Opinnäytetyö (AMK)

Tietotekniikan koulutusohjelma

Mediatekniikka

2011

Joni Toiviainen

GRAFIIKKAMOOTTORIN POHJA-ARKKITEHTUURIN SUUNNITTELU JA TOTEUTUS

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikan koulutusohjelma | Mediatekniikka

2011 | 48

Ohjaajat: Ins. (AMK) Keijo Leinonen, KTT Reetta Raitoharju

Joni Toiviainen

GRAFIIKKAMOOTTORIN POHJA-ARKKITEHTUURIN SUUNNITTELU JA TOTEUTUS

Tässä opinnäytetyössä suunniteltiin ja toteutettiin nykyaikaiseen teknologiaan perustuvan grafiikkamoottorin pohja-arkkitehtuuri. Pohja-arkkitehtuuri rakennettiin ohjelmalisäke-arkkitehtuurin mukaisesti, jolloin sen rakenteesta saatiin dynaaminen.

Tekniikkoina hyödynnettiin nykyaikaisten vapaan lähdekoodin grafiikkamoottorien käyttämiä tekniikoita yhdistettynä sovelluskehityksessä käytettäviin suunnittelumalleihin. Grafiikka-moottorin toiminta ei ole sitoutunut noudattamaan vain ennalta määritettyjä tekniikoita, vaan arkkitehtuuri antaa myös mahdollisuuden soveltaa käytettyjä tekniikoita tarpeen mukaan.

Lopputuloksena saatu pohja-arkkitehtuuri sallii grafiikkamoottorin laajentamisen erityyppisten ohjelmalisäkkeiden avulla. Ohjelmalisäkkeet mahdollistavat tulevaisuuden grafiikka-ohjelmointirajapintojen tuonnin järjestelmään sekä mahdollisuuden laajentaa järjestelmän resurssitukea.

Järjestelmän päätietorakenne on näkymägraafi. Näkymägraafin solmut luodaan keskitetyllä tehdasmenetelmällä, joka mahdollistaa uusien solmujen luonnin järjestelmään helposti.

Pohja-arkkitehtuuri saatiin rakennettua vaatimusmäärittelyn mukaisesti, ja se on siirretty jatkokehitykseen.

ASIASANAT:

grafiikkamoottori, ohjelmistokehitys, ohjelmointi, ohjelmistoarkkitehtuuri

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information Technology | Digital Media

2011 | 48

Instructors: Keijo Leinonen, B. Eng., Reetta Raitoharju, D. Sc. (Econ. & Bus. Adm.), Principal Lecturer

Joni Toiviainen

DESIGN AND IMPLEMENTATION OF A GRAPHIC ENGINE BASE ARCHITECTURE

The subject of this thesis was to design and implement a graphic engine architecture which utilizes modern technologies used in current graphic engines. Base architecture was built by using plug-in architecture principles to ensure that the system would have dynamic expansion capabilities.

Base architecture uses a combination of technologies used in open source graphic engines and design patterns which are common in software development. Technologies used in the base architecture design can be further developed in the actual graphic engine implementation.

The built base architecture allows the use of plug-ins. Plug-in system allows the base architecture to be expanded via external files that can improve the rendering and resource support of the system.

Main data structure of the architecture is a scene graph. Scene graph nodes are created by using an advanced factory method, which can easily be extended with new node types.

The design and implementation of the base architecture was a success, and the engine is now in further development.

KEYWORDS:

Graphic engine, software development, software, software architecture

SISÄLTÖ

KUVAT

SANASTO

CPU	Tietokoneen prosessori (Central Processing Unit), jonka avulla suoritetaan järjestelmän laskentaoperaatiot.
DirectX	Microsoftin kehittämä ohjelmointirajapinta, joka sisältää rajapinnat esimerkiksi ääni-, grafiikka-, verkko- ja ohjauslaitteiden hallintaan.
Dynaaminen kirjasto	Ohjelmakoodivarasto, joka pitää sisällään erilaisia tietokoneohjelman rakentamisessa käytettäviä rutiineja. Mahdollista käyttää ohjelmassa erillisenä tiedostona lataamalla kirjasto ohjelman käyttöön ajon aikana tai käynnistyksen yhteydessä.
GPU	Näytönohjaimen grafiikkaprosessori (Graphic Processing Unit), joka suorittaa kuvanmuodostusvaiheen laskentaoperaatiot.
Piirtoputki	Näytönohjaimen grafiikkaprosessorin järjestelmä, joka vastaa geometrian muuttamiseksi rasteroiduksi kuvaksi.
Rasterointi	Menetelmä, jossa grafiikkaprimitiivit muutetaan kuvapisteiksi.
Rtti	C++:n ajonaikainen tyyppitunnistusjärjestelmä (Run-Time Type Information).
Staattinen kirjasto	Ohjelmakoodivarasto, joka pitää sisällään erilaisia tietokoneohjelman rakentamisessa käytettäviä rutiineja. Kirjastosta tarvittavat osat sisällytetään sitä käyttävään ohjelmakoodiin käännösvaiheessa.
Suunnittelumalli	Yleinen menetelmä, jota hyödynnetään järjestelmän suunnittelun ja toteutuksen parissa.
Verteksi	Geometrinen kulmapiste, joka sisältää vähintään tiedon sijainnista x-, y- ja z-koordinaatistossa.
Varjostin	Grafiikkaprosessorilla suoritettava ohjelma, joka on kirjoitettu erillisellä varjostinohjelmointikielellä.
Xml	Merkintäkieli, jota käytetään tiedon kuvailuun (Extensible Markup Language).

1 JOHDANTO

Nykyajan peliteollisuus on kasvanut jo elokuva- ja musiikkiteollisuuden rinnalle yhdeksi suurimmista media-aloista. Osa nykyaikaisista peleistä voidaan jo luokitella interaktiivisuutta sisältäviksi elokuviksi niiden sisältämien draamallisen ja visuaalisen sisällön ansiosta. Osa peleistä on pyritty viemään mahdollisimman lähelle realismin illuusiota tuottamalla peliin erittäin näyttävää grafiikkaa. Näyttävän grafiikan kanssa on kuitenkin huomiotava laitetason rajallinen suorituskyky.

Vaikka järjestelmän laitteiston suorituskyky on rajallinen, voidaan grafiikan muodostukseen käytettäviä laskenta-aikoja parantaa kehittämällä tehokkaita tietorakenteita ja algoritmeja. Interaktiivisen ja realismin illuusiota tavoittelevan pelin eniten laskentatehoa vaativa yksikkö on grafiikan muodostuksesta vastaava grafiikkamoottori. Nykyajan pelien grafiikkamoottoreissa hyödyntävät jo niin tehokkaita algoritmeja, että niitä käytetään hyödyksi myös terveysalalla esimerkiksi sydänleikkausten harjoittelussa [1].

Tämän työn tarkoituksena on tutkia nykyaikaisten grafiikkamoottorien pohja-arkkitehtuuria vertailemalla olemassa olevien grafiikkamoottorien pohja-arkkitehtuureja keskenään sekä tutkimalla alan kirjallisuutta. Tutkimustyön jälkeen sovelletaan tutkimustuloksia suunnittelemalla ja rakentamalla oma grafiikkamoottorin pohja-arkkitehtuuri. Pohja-arkkitehtuurin rakentamisessa tulee erityisesti huomioida sen laajennettavuus jatkokehityksen kannalta.

Työssä kehitettävä grafiikkamoottori nimeltä Tartarus3D toteutetaan neljän henkilön muodostamalle FantasyCraft-pelikehitysryhmälle. Kun grafiikka-moottorin pohja-arkkitehtuuri on saatu toteutettua, siirtyy tämä työ suoraan jatkokehitykseen FantasyCraftin tämänhetkisen peliprojektin pariin.

2 GRAFIIKKAMOOTTORI

Grafiikkamoottori on tietokonesovelluksen osa, jonka avulla on mahdollista esittää tietoa visuaalisesti. Visuaalisen esityksen rakentamiseksi grafiikkamoottori lähettää sovelluksesta tietoa järjestelmän näytönohjaimelle, joka puolestaan muuttaa tiedon näytölle esitettävään muotoon.

Nykyaikaiset grafiikkamoottorit muodostavat kuvan käyttämällä näytönohjaimen piirtoputkea. Piirtoputkeen syötettävä tietovirta pitää sisällään geometrisia määrityksiä, joista muodostetaan geometrisia muotoja piirtoputken verteksi- ja geometriajärjestelmissä. Lopuksi piirtoputki rasteroi geometriamäärityksestä kuvan, joka voidaan esittää visuaalisesti näytölle tai vaihtoehtoisesti käyttää osana muiden geometrioiden pinnoitteita. [2]

Geometriatiedon lisäksi grafiikkamoottori käsittelee erilaisia kuvan-muodostukseen liittyviä operaatioita, kuten geometrian pinnoitteeseen vaikuttavia efektejä, esitettävän kuvan animointia sekä geometriaobjektien törmäystarkistuksia.

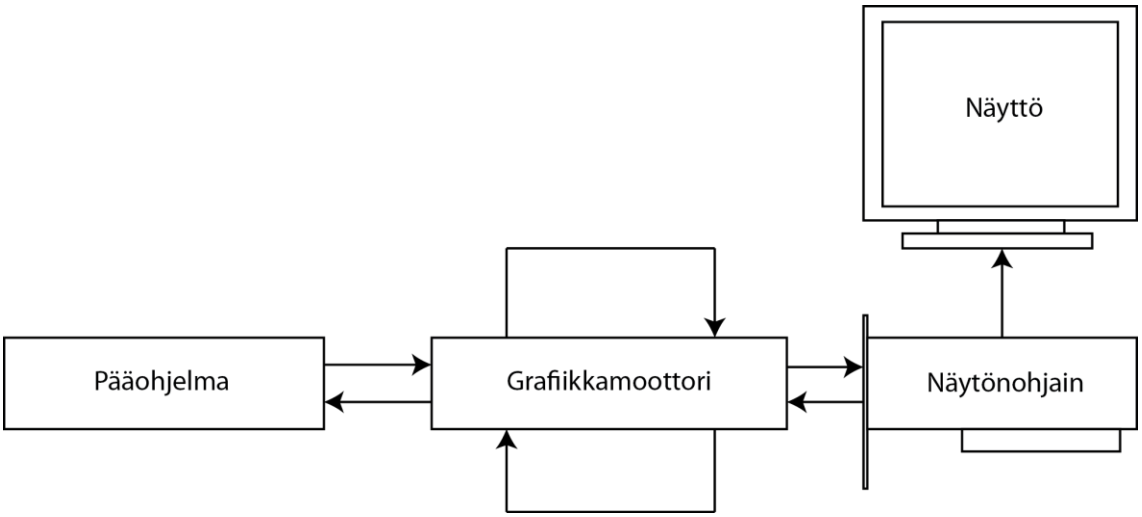
Grafiikkamoottorissa kulkee suuria tietovirtoja, joiden ohjaamisessa käytetään kehittyneitä tietorakenteita ja algoritmeja. Yleisimmin käytetty tietorakenne näkymägraafi muodostuu graafitietorakenteesta [3].

2.1 Päätietovirrat

Grafiikkamoottorien pohjatoiminta perustuu tietovirtojen hallintaan kuvanmuodostusjärjestelmän ja näkymägraafin välillä. Kuvanmuodostus-järjestelmän ja näkymägraafin välillä kulkevan tietovirran lisäksi grafiikkamoottorit käyttävät yleensä ulkopuolisia tietovirtoja tiedonlähteinään. Ulkopuolisia tietovirtoja voivat olla esimerkiksi kuvat ja geometriamallit, joita käytetään kuvanmuodostusoperaatioissa. Päätietovirrat on mahdollista jakaa kolmeen pääkategoriaan. Yksi tietovirta käsittelee grafiikkamoottoria käyttävän

ohjelman kanssa käsiteltävää ja resurssien muodostuksesta vastaavaa tietovirtaa, toinen käsittelee grafiikkamoottorin sisäisesti toimivaa tietovirtaa ja kolmas käsittelee kuvanmuodostuksen tietovirtaa. [3]

Kuva 1 havainnollistaa, miten näytönohjaimen kuvanmuodostusmenetelmien hallinta poikkeaa muista tietovirroista huomattavasti, sillä se hyödyntää myös GPU:n laskentatehoa. Nykyajan näytönohjaimien GPU:ssa on huomattavasti enemmän laskentatehoa kuin tietokoneen CPU:ssa, joka tekee niistä tehokkaan apuvälineen laskentaoperaatioiden suorittamisessa. [4, 5]



Kuva 1. Grafiikkamoottorin tietovirrat.

2.2 Ohjelmointirajapinta

Grafiikkamoottori käyttää näytönohjaimen kanssa keskustellessaan apuna grafiikkaohjelmointiin kehitettyä ohjelmointirajapintaa. Grafiikkaohjelmointi-rajapinta sisältää funktioita, rakenteita ja apukomentoja, joita käytetään näytönohjaimen toiminnan hallitsemiseen. Kaksi eniten käytettyä ohjelmointirajapintaa ovat Khronos Groupin kehittämä OpenGL sekä Microsoftin Direct3D, joka on 3d-grafiikkan muodostamiseen keskittynyt osa Microsoftin DirectX-järjestelmässä. [6, 7]

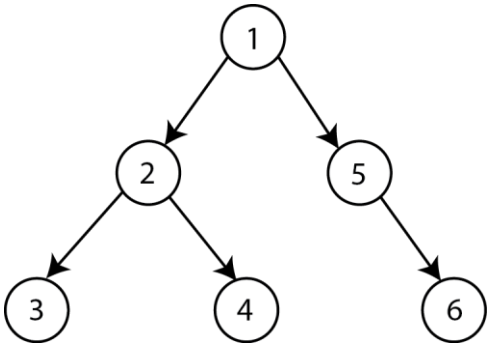
Grafiikkaohjelmointirajapinnan käyttö edellyttää rajapinnan kirjaston linkityksen sitä käyttävän ohjelman kanssa. Linkityksen jälkeen ohjelmasta tulee sidonnainen grafiikkaohjelmointirajapinnan kanssa, jolloin aina ohjelman käynnistyksen yhteydessä ohjelma tarkistaa, että tietokonejärjestelmästä löytyy sen hyödyntämä grafiikkaohjelmointirajapinta asennettuna. Sidonnaisuuden lisäksi grafiikkaohjelmointirajapinnat vaativat myös tuen näytönohjaimelta toimiakseen oikealla tavalla. [8]

2.3 Näkymägraafi

Grafiikkamoottorin näkymägraafi toimii moottorin päätietorakenteena. Näkymägraafi sisältää grafiikkamoottorin kuvanmuodostukseen tarvittavat määritykset ja objektit. Graafi muodostuu solmuista ja kaarista, jotka muodostavat keskenään tietoverkon. Graafi muistuttaa puutietorakennetta, mutta sillä poikkeuksella, että graafin solmuilla voi olla useampia vanhempia. [9]

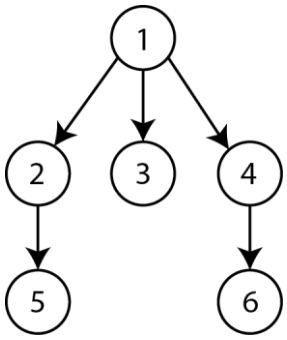
Graafin tehokkuus perustuu sen läpikäyntimenetelmiin. Kuvanmuodostukselle lähetettävä tietovirta muodostetaan käymällä läpi kaikki näkymägraafin solmut samalla päivittäen kuvanmuodostustilaa. Yksi yleinen tapa on hyödyntää syvyyden mukaan läpikäyntiä, joka toimii samalla periaatteella kuin puu-tietorakenteen läpikäynti. [10]

Kuva 2 havainnollistaa syvyyden mukaan läpikäynnin, jossa valitaan aluksi solmu, josta lähdetään liikkeelle. Solmun lapsisolmut lisätään listaan ja valitaan ensimmäinen lapsisolmu käsiteltäväksi. Käsiteltävän lapsisolmun lapsisolmut listataan ja ne läpikäydään. Mikäli lapsisolmulla ei ole lapsisolmuja, niin siirrytään takaisin ylöspäin ja valitaan seuraava lapsisolmu käsittelyyn. Algoritmi on helppo toteuttaa rekursion avulla.



Kuva 2. Graafin solmujen läpikäyntijärjestys syvyyden mukaan.

Kuva 3 esittää toisen läpikäyntimenetelmän, joka on leveyden mukaan läpikäynti, jossa aloituskohtaa lähimpänä olevien solmujen käsittely toteutetaan ensin. Läpikäynnin aikana läpikäytyjen solmujen lapsisolmut kerätään jono-tietorakenteeseen odottamaan seuraavaa kierrosta. Kun kaikki ensimmäisen kierroksen solmut on läpikäyty, voidaan aloittaa toisen kierroksen solmujen läpikäynti. Läpikäyntiä jatketaan kunnes kaikki solmut on käyty läpi.



Kuva 3. Graafin solmujen läpikäyntijärjestys leveyden mukaan.

3 SUUNNITTELU

3.1 Vaatimusmäärittely

FantasyCraftin peliprojektissa käytettävälle grafiikkamoottorille on asetettu eräitä ennalta määritettyjä vaatimuksia, jotka tulee ottaa huomioon myös pohja-arkkitehtuurin rakennusvaiheessa.

Dynaaminen rakenne

Dynaamisen pohja-arkkitehtuurin saavuttaminen on korkeimmalla prioriteettitasolla. Dynaamisuuden avulla pyritään mahdollistamaan järjestelmän vaivaton laajentaminen jatkokehitysvaiheessa. Joustavuus halutaan mukaan järjestelmään sekä ulko- että sisäpuolisesti.

Tietovirtojen tasapainotus

Tietovirtojen käsittely tulee toteuttaa pohja-arkkitehtuurin lohkojen välillä tasapainoisesti. Arkkitehtuurissa tulee pyrkiä välttämään saman tietovirran uudelleenmuodostamista saman sisällön kanssa. Tasapainotetut tietovirrat mahdollistavat grafiikkamoottorin tasapainoisen rasituksen GPU:n ja CPU:n välille.

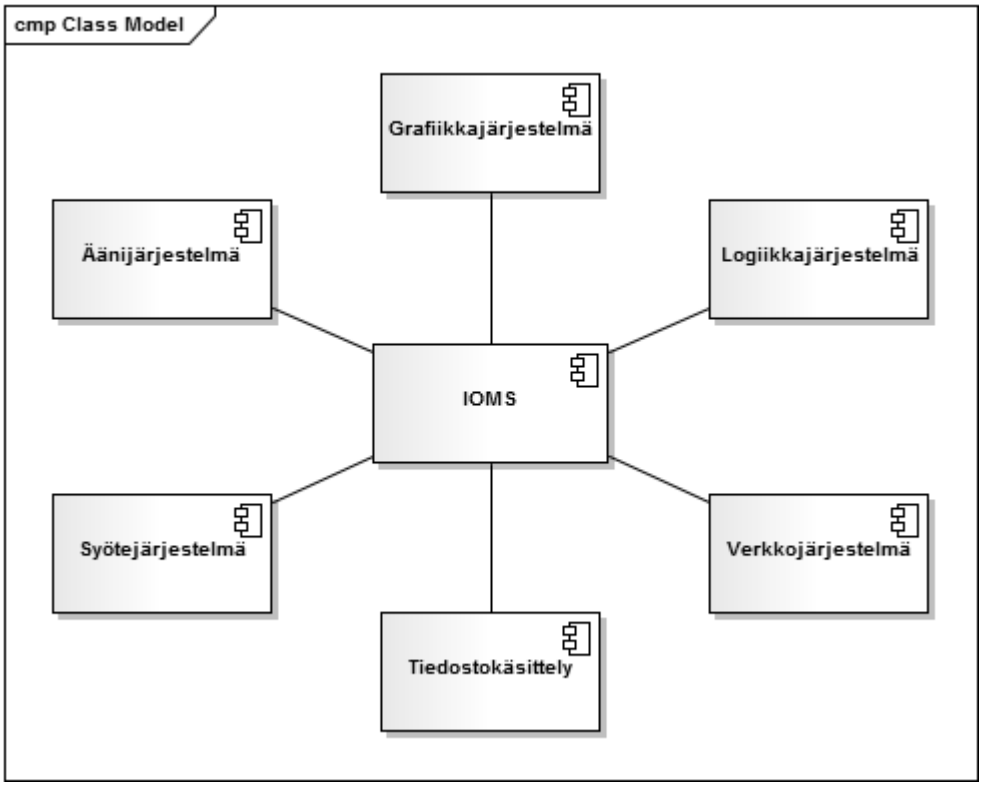
Muistinkäytön järkevä hallinta

Ohjelmointikieleksi tulee valita järjestelmän muistinkäsittelyn mahdollistava ohjelmointikieli. Muistinkäsittelyä tarvitaan grafiikkamoottorin muistinkäsittelyn optimoinnissa, kun tietovirtojen suuruudet kasvavat. Lisäksi ohjelmointikielen tulee tukea olio-ohjelmointia.

3.2 Kohdepelimoottorin arkkitehtuuri

Kohdepelimoottorin arkkitehtuuri on jalostettu tietokeskeisestä järjestelmä-ohjelmistoarkkitehtuurista. Pelimoottorin pääjärjestelmänä toimii IOMS (Intelligent Object Management System), joka käsittelee eri järjestelmien väliset yhteydet ja hoitaa pelimoottorin objektien korkean tason hallinnan. IOMS ei kuitenkaan hoida resurssien varastointia tai alajärjestelmän sisäisiä tietovirtoja.

Toteutettavan grafiikkamoottorin tulee toimia itsenäisenä järjestelmänä kuvan 4 havainnollistaman arkkitehtuurin mukaisesti. Grafiikkajärjestelmä eli grafiikka-moottori ei luo suoraa relaatiota muihin pelimoottorin alajärjestelmiin vaan suorittaa keskustelun käyttäen apuna IOMS-järjestelmää.



Kuva 4. Tietokeskeinen IOMS-pelimoottoriarkkitehtuuri.

Pelimoottorin alajärjestelmät ovat dynaamisia kirjastoja, joiden avulla on mahdollista luoda yksilökohtaisia päivityksiä järjestelmiin. Tartarus3D luodaan päämoottorin arkkitehtuurin mukaisesti dynaamiseksi kirjastoksi.

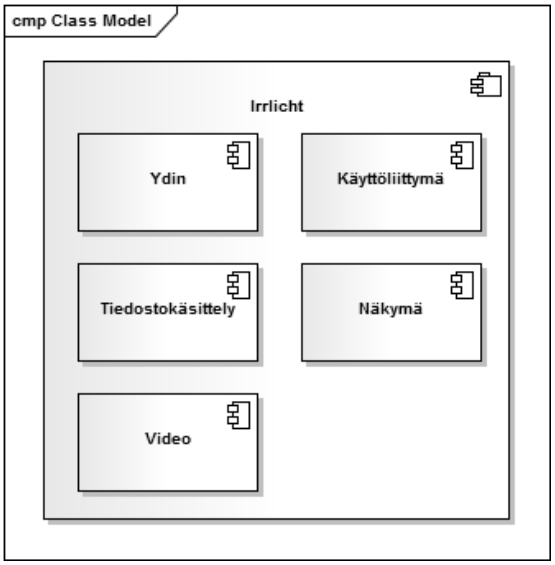
3.3 Arkkitehtuurin lohkot

Arkkitehtuuri tulee jakaa lohkoihin, jotta sen kehittäminen ja hallinnointi on sujuvaa. Aluksi tulee perehtyä millä tavoin olemassa olevien grafiikkamoottorien pohja-arkkitehtuurin lohkotus on muodostettu. Tutkimuksessa käytetään apuna sekä grafiikkamoottorien lähdekoodeja, että dokumentaatioita. Tutkimustulosten jälkeen on mahdollista soveltaa kerättyä tietoa oman pohja-arkkitehtuurin rakentamiseen.

3.3.1 Irrlichtin lohkot

Irrlicht on vapaan lähdekoodin grafiikkamoottori, jonka kehitystavoitteita ovat yksinkertaisuus, helppokäyttöisyys ja nopeus. Ohjelmakoodi on kirjoitettu käyttämällä C++-ohjelmointikieltä, mutta kehittäjät ovat luoneet tuen myös muille kielille. [11, 12]

Irrlichtin arkkitehtuuri on jaettu kuvan 5 mukaisesti viiteen lohkoon, jotka muodostuvat järjestelmän päälohkosta, käyttöliittymälohkosta, tiedostolohkosta, näkymälohkosta ja kuvanmuodostuslohkosta.



Kuva 5. Irrlicht-grafiikkamoottorin päälohkot.

Ydinlohko sisältää järjestelmälle yhteisiä objekteja. Yhteisiä objekteja ovat esimerkiksi matemaattiset objektit sekä järjestelmässä käytettävät tietorakenteet. Suurin osa ydinosan objekteista on rakennettu malliluokkina, joka mahdollistaa luokkien käytön geneerisen ohjelmoinnin avulla.

Käyttöliittymälohko sisältää käyttöliittymäkomponenttien luokat, sekä niissä tapahtuvien toimintojen hallintaan kehitetyt luokat. Tiedostokäsittelylohko käsittelee grafiikkamoottorin tiedostojen luku- ja kirjoitusoperaatioita. Irrlichtiin on rakennettu tuki myös tiedostopakettien purku- ja luontioperaatioille. Lisäksi järjestelmästä löytyy tuki xml-tiedostojen käsittelylle.

Näkymälohko on luokkamäärältään suurin lohko koko arkkitehtuurissa. Se pitää sisällään kaikki näkymän solmutyypit, sekä niiden hallintaan tarvittavat luokat. Näkymälohko sisältää myös verteksien puskurointiin tarvittavat luokat, joiden avulla tieto välitetään videolohkolle kuvanmuodostukseen.

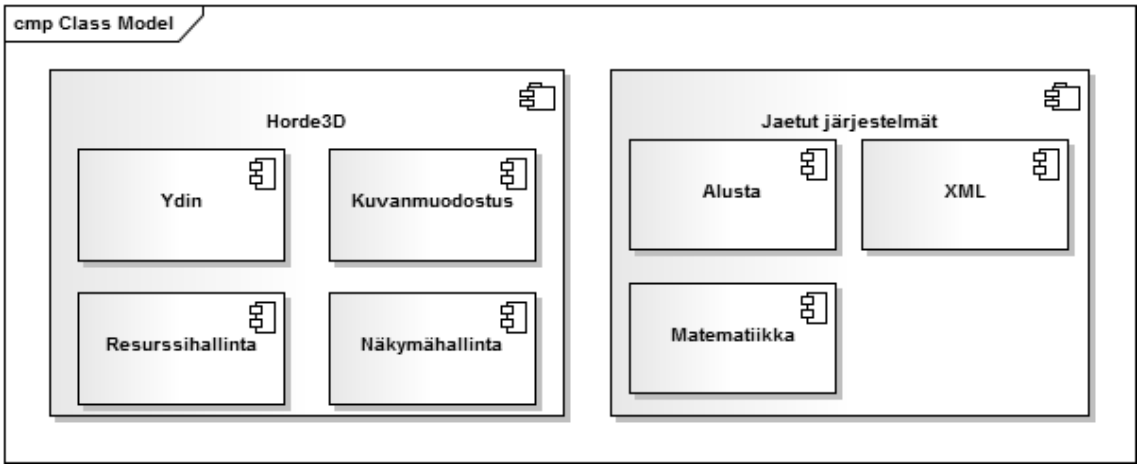
Videolohko toimii Irrlichtin kuvanmuodostusjärjestelmänä. Sen avulla käsitellään kuvanmuodostuksessa tarvittavia pinnoitteita ja GPU:lle määritettäviä ohjelmia. Videolohko toimii yhteistyössä näytönohjaimen kanssa käyttäen grafiikka-ohjelmointirajapintaa. Lohkoon on myös rakennettu mahdollisuus järjestelmän näytönohjaimen tietojen ja näytön ominaisuuksien luetteloinnille.

Irrlichtin kuvanmuodostusjärjestelmä tukee sekä OpenGL että Direct3D rajapintoja. Niiden lisäksi kuvanmuodostukseen on mahdollista käyttää ohjelmistopohjaista kuvanmuodostusjärjestelmää.

3.3.2 Horde3D:n lohkot

Horde3D on Nicolas Schulzin kehittämä vapaan lähdekoodin grafiikkamoottori. Se on kirjoitettu hyvin pienellä määrällä lähdekoodia, mutta yltää siitä huolimatta nykyaikaiseen grafiikkamoottoreilta odotettavalle tasolle. Horde3D:n kuvanmuodostusjärjestelmä toimii OpenGL-rajapinnan avulla ja se on kirjoitettu käyttämällä C++-ohjelmointikieltä. [13]

Kuva 6 havainnollistaa sitä, kuinka Horde3D:n arkkitehtuuri on jaettu 4 päälohkoon ja lohkot hyödyntävät myös jaettuja järjestelmiä. Lohkojen erottelu toisistaan on hyvin heikosti havaittavissa lähdekoodissa, sillä lähdekoodin määrä on pyritty tiivistämään mahdollisimman pieneen tilaan.



Kuva 6. Horde3D:n arkkitehtuurin päälohkot.

Ydinlohko pitää sisällään Horde3D:n ajastuksen, lokimerkintöjen ja ohjelmalisäkkeiden hallintaan tarkoitettuja operaatioita. Ydinlohko alustaa muut lohkot Horde3D:n käynnistyksen yhteydessä. Alustuksessa resurssilohkon ja näkymälohkon objektien luontiyksiköille rekisteröidään käytettävissä olevat objektityypit.

Resurssijärjestelmä pitää sisällään resurssien lataukseen tarvittavat operaatiot ja objektiluokat. Horde3D:n resurssit ladataan käyttämällä XML-määrityksiä. Määritykset syötetään hallintayksiköille, joka lataa resurssit XML-tiedostoon määritettyjen parametrien mukaisesti. Horde3D:n resurssijärjestelmä kattaa hyvin laajasti koko grafiikkamoottorin elementit mukaan lukien varjostimet.

Horde3D:n näkymähallintalohko on keskittynyt spatiaaliseen graafiin ja siihen liitettäviin solmuihin. Solmut rakennetaan käyttämällä solmun tyyppikohtaista *Factory Method* -suunnittelumallin mukaista rakennusfunktiota [14].

Kuvanmuodostuslohko on yhteydessä OpenGL-rajapinnan avulla näytönohjaimeen. Lohkon avulla muodostetaan GPU:lla ajettavat ohjelmat, sekä määritetään piirtoputken tilavaihdokset.

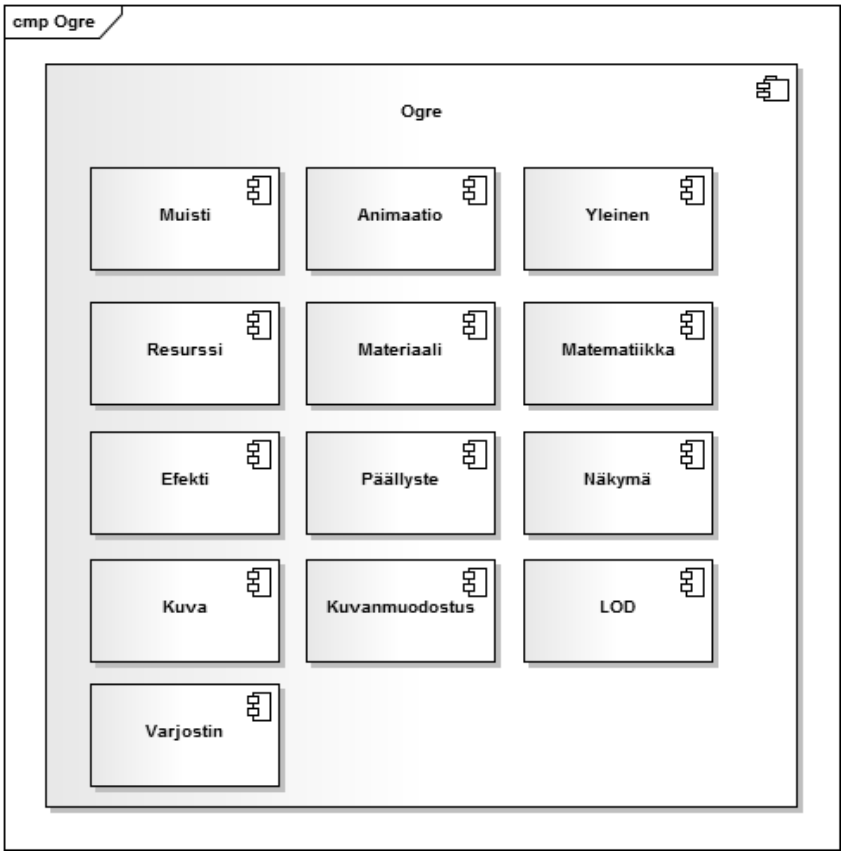
Horde3D:ssa on pääjärjestelmän lisäksi jaettuja järjestelmiä. Jaetut järjestelmät ovat käytössä päämoottorissa sekä ulkopuolisissa työkaluissa. XML-järjestelmä on Horde3D:n tiedostojen latausjärjestelmän tärkeä osa, jonka avulla käyttäjän luomia XML-tiedostoja voidaan tulkita resurssien muodostuksessa. Matematiikkajärjestelmä pitää sisällään geometrisessa matematiikassa tarvittavia rakenteita ja funktioita. Alustajärjestelmä käsittelee alusta- ja kääntäjäkohtaisia määrityksiä.

Horde3D hyödyntää ulkopuolista kääntäjää 3d-mallien muuttamiseksi sopivaan muotoon. Kehitysympäristön mukana tulee ohjelma nimeltä Collada Converter, jonka avulla on mahdollista kääntää Collada-tiedostomuodossa olevia tiedostoja Horde3D:lle.

3.3.3 Ogren lohkot

Ogre (Object-oriented Rendering Engine) on Torus Knot Software Ltd:n kehittämä suuri ja monipuolinen vapaan lähdekoodin grafiikkamoottori. Ogren kehitystavoitteena on monipuolisuus ja tehokkuus, joka antaa käyttäjälle mahdollisuuden laajentaa grafiikkamoottoria erilaisten ohjelmalisäkkeiden avulla. [15]

Kuva 7 havainnollistaa, miten Ogren arkkitehtuuri on jaettu useaan osaan. Lähdekoodin määrä on moninkertainen Irrlichtin ja Horde3D:n lähdekoodin määrään. Rakenteen avulla on pyritty rakentamaan erittäin monipuolinen ja optimoitu järjestelmäkokonaisuus. Lohkojen sisällä ovat pienimmätkin toiminnallisuudet luotu omiksi luokikseen, jotta niiden toimintaa on mahdollista muokata monipuolisesti.



Kuva 7. Ogren arkkitehtuurin päälohkot. [16]

Muistilohkossa on optimoituja muistihallinnan menetelmiä, joita ovat esimerkiksi optimoitu muistin vapautus ja varaaminen. Muistinkäsittely tehdään pääsääntöisesti käyttäen Ogren omia menetelmiä. Ogren arkkitehtuurirakenteessa on selvästi havaittavissa, että järjestelmä on hyvin pitkälle optimoitu, sillä pienimmätkin operaatiot on purettu erillisiksi osiksi.

Ogren yleisessä lohkossa on suunnittelumallien, poikkeusten ja yleisten työkalujen toteutukset. Lohko sisältää muille järjestelmille tarpeellisia apurakenteita kehitystyön helpottamiseksi.

Muut lohkot sisältävät nimensä puolesta oleellisia luokkia ja rakenteita, jotka on jaettu hyvin pieniin osiin. Ogren hienoin ominaisuus on siihen toteutettu ohjelmalisäkejärjestelmä, jonka avulla on mahdollista tuoda järjestelmään myös uusia tietorakenteita.

3.3.4 Tartarus3D:n lohkot

Grafiikkamoottoreilla on aina vähintään tietyt lohkot niiden rakenteissa. Lohkot ovat tiedostokäsittely, näkymähallinta ja kuvanmuodostus. Irrlichtin sisältämä käyttöliittymälohko vaikuttaa harvinaiselta lohkolta grafiikkamoottorin arkkitehtuurissa. Käyttöliittymäkomponenttien sisällyttäminen grafiikkamoottoriin antaa mahdollisuuden optimoida komponenttien toimintaa grafiikkamoottorin perusmekaniikan kanssa. Tartarus3D:hen ei rakenneta omaa käyttöliittymäkomponenttilohkoa, koska käyttöliittymäkomponentit tullaan rakentamaan IOMS-järjestelmässä, jotta interaktiivisten käyttöliittymä-komponenttien hallinta voidaan jakaa logiikka-, grafiikka- ja syötejärjestelmien kesken.

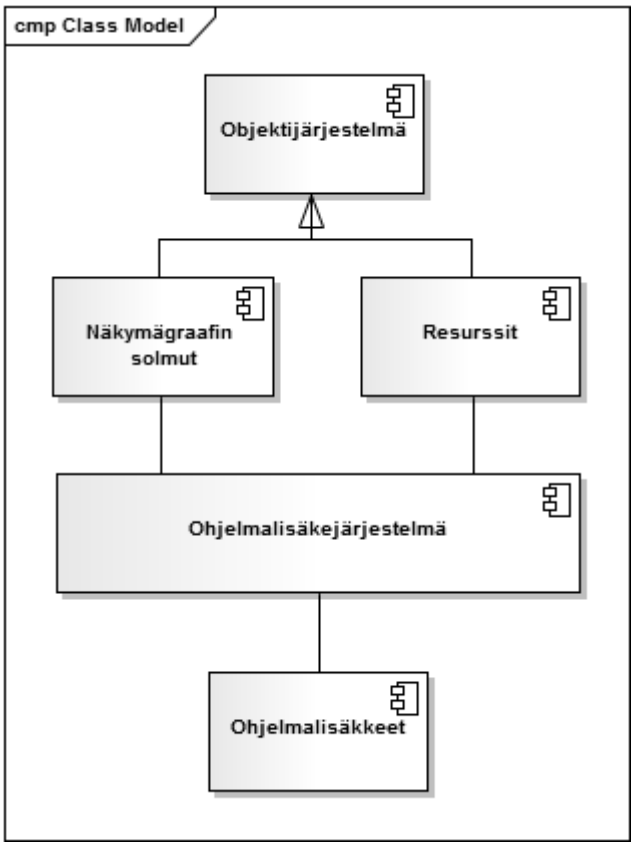
Irrlichtin tiedostokäsittelylohko on toiminnaltaan sama kuin Ogren ja Horde3D:n resurssilohkot. Resurssien hallinta on tärkeä lohko, sillä sen avulla hallinnoidaan ulkopuolista tietovirtaa. Lisäksi resurssien hallinta toimii osittaisena siltana sisäisen tietovirran ja kuvanmuodostukseen käytettävän tietovirran kanssa.

Arkkitehtuurin monipuolisuuden parantamiseksi Tartarus3D:n arkkitehtuurin pohjarakenne toteutetaan ohjelmalisäkearkkitehtuurina kuten Ogressa. Ohjelmalisäkearkkitehtuurin avulla on mahdollista suorittaa järjestelmän dynaaminen laajentaminen ilman muutoksia grafiikkamoottorin pääarkkitehtuuriin.

Vaihtoehtoinen tapa toteuttaa ohjelman dynaaminen rakenne olisi käyttää erilaisia integroituja tietorakenteita, joiden avulla olisi mahdollista rekisteröidä uusia objekteja järjestelmän käyttöön. Rekisteröinti olisi mahdollista toteuttaa ohjelman sisäinen hallinnan kautta käyttämällä dynaamista objektikehitystä. Dynaamisen objektikehityksen avulla järjestelmän dynaaminen laajentaminen vaatimusmäärityksen mukaan olisi mahdollista, mutta se ei olisi yhtä sulavaa kuin ohjelmalisäkkeiden avulla. Ohjelmalisäkkeiden suurin etu on niiden päivitettävyys päämoottorin ulkopuolella, ilman muutoksia itse päämoottoriin.

Horde3D:n käyttämä XML-perustainen resurssimääritys ei ole toimiva ratkaisu Tartarus3D:n arkkitehtuurin kanssa, sillä Tartarus3D:n resurssimääritykset tehdään IOMS-järjestelmän avulla. Resurssilohko ja näkymälohko rakennetaan omiksi lohkoiksi, mutta niiden objektit voidaan periyttää niitä yhdistävästä objektijärjestelmästä.

Tartarus3D:n ohjelmalisäkepohjainen arkkitehtuuri soveltaa Ogren, Horde3D:n ja Irrlichtin menetelmiä kuvan 8 mukaisesti. Ohjelmalisäkejärjestelmä toimii rajapintana ohjelmalisäkkeiden ja objektijärjestelmästä periytyvien solmujen ja resurssien välillä. Ohjelmalisäkejärjestelmästä on mahdollista tehdä niin laaja, että Tartarus3D muuttuu täysin ohjelmalisäkepohjaiseksi järjestelmäksi. Täysin ohjelmalisäkepohjaisessa järjestelmässä järjestelmän kaikkia osia on mahdollista laajentaa dynaamisesti.



Kuva 8. Tartarus3D:n pohja-arkkitehtuuri.

Kuvanmuodostuslohkoa ei toteuteta suoraan järjestelmään, vaan se toteutetaan ohjelmalisäkkeenä. Tällä tavoin näkymähallintalohkojen grafiikkaohjelmointirajapintakohtaisten kutsut on mahdollista poistaa pääarkkitehtuurista.

Vaihtoehtoisesti kuvanmuodostuslohko olisi mahdollista toteuttaa hyödyntämällä grafiikkaohjelmointirajapintaa suoraan grafiikkamoottorin päälohkossa, jolloin näytönohjaimen käskytys tehtäisiin suoraan grafiikkamoottorin ytimessä. Tällöin ei tarvittaisi ulkopuolisia ohjelmalisäkkeitä kuvanmuodostukseen ja samalla olisi mahdollista helpottaa pohja-arkkitehtuurin rakentamista käyttämällä grafiikkaohjelmointirajapintojen omia matematiikkakirjastoja avustamaan kehitystyötä. Vaihtoehtoinen tapa ei kuitenkaan ole hyvä menetelmä järjestelmän jatkokehityksen kannalta, sillä järjestelmän elinkaari olisi silloin rajoittunut vain yhteen grafiikkaohjelmointirajapintaan. Tulevaisuuden grafiikkaohjelmointirajapintojen lisääminen järjestelmään vaikeutuisi huomattavasti, koska lisääminen pitäisi tehdä suoraan grafiikkamoottorin päälohkoon. Järjestelmä on selkeämmin ja joustavammin laajennettavissa kun kuvanmuodostuslohko toimii ohjelmalisäkkeenä.

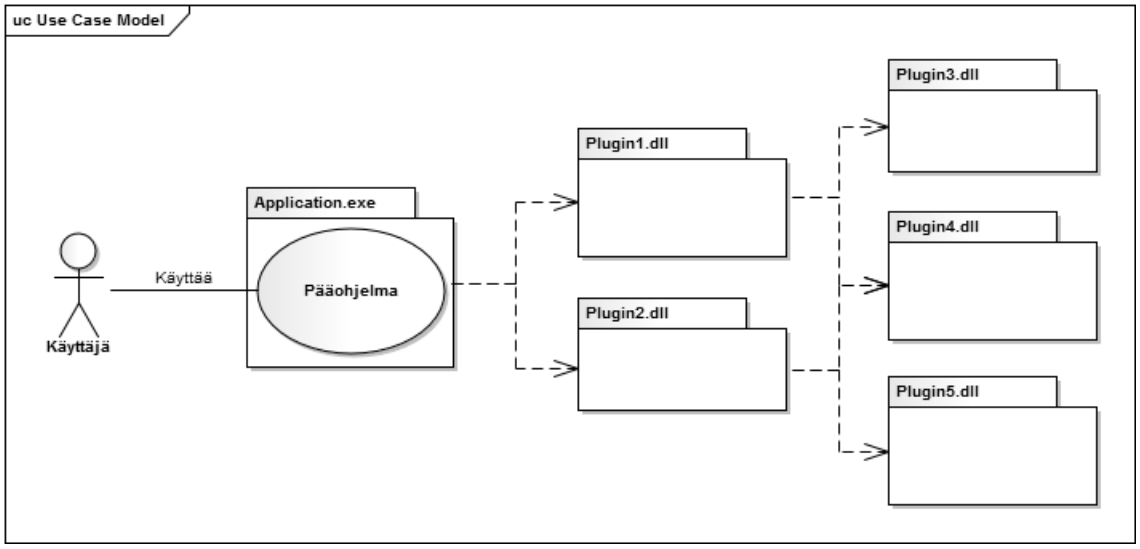
Tartarus3D:n lokijärjestelmänä käytetään FantasyCraftin kehittämää Infog-lokijärjestelmää, joka on mahdollista ladata ajon aikana grafiikkamoottorin käyttöön. Lokijärjestelmä on toteutettu dynaamisena linkkikirjastona.

3.4 Ohjelmalisäkearkkitehtuuri

Ohjelmalisäkearkkitehtuuri mahdollistaa ulkopuolisten järjestelmien tuonnin pääjärjestelmän käyttöön. Ohjelmalisäkepohjainen arkkitehtuuri mahdollistaa myös dynaamisten kirjastojen välisen verkon luomisen. Verkon avulla on mahdollista luoda ohjelmakokonaisuus, jonka toimintaa on mahdollista muokata lisäämällä tai poistamalla ohjelmalisäkkeitä.

Kun dynaamiset kirjastot muodostavat kuvan 9 mukaisen verkon, voivat ne käyttää apunaan myös yhteisiä dynaamisia kirjastoja. Kuvan esimerkin tavalla on mahdollista käyttää Plugin4.dll-kirjastoa yhteisenä kirjastona Plugin1.dll- tai

Plugin2.dll-kirjastojen kanssa. Esimerkki yleisestä yhteisestä kirjastosta on lokijärjestelmä.



Kuva 9. Dynaamisten kirjastojen muodostama verkko.

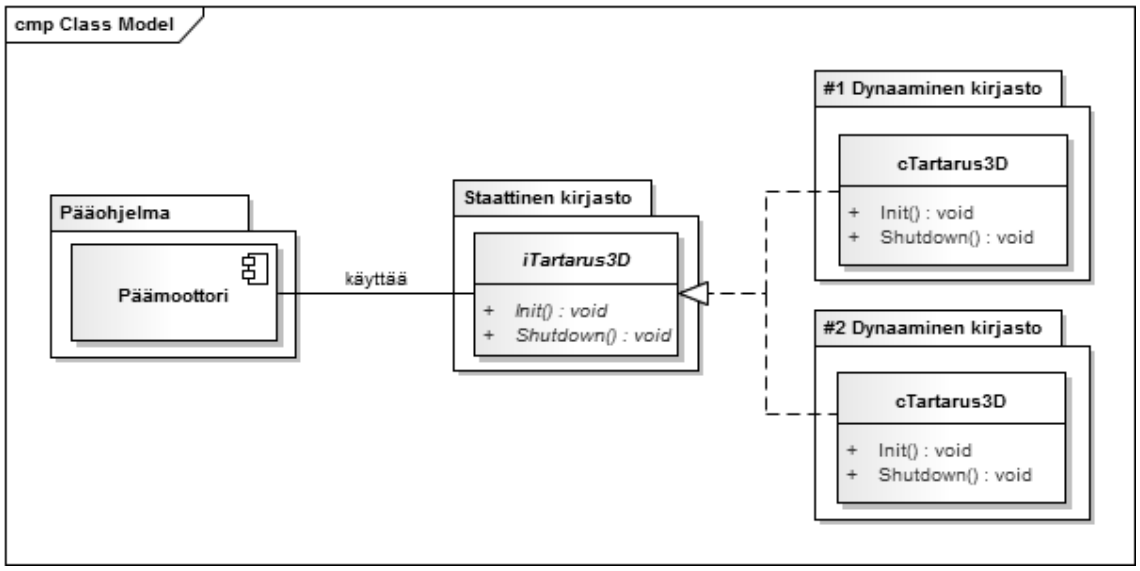
Ohjelmalisäkkeiden lisääminen järjestelmän arkkitehtuuriin tulee tehdä niiden hallintaan keskittyneen yksikön avulla. Hallintayksikkö suorittaa lisäkkeiden vapautuksen ja tallennuksen grafiikkamoottorin ohjelmalisäketietokantaan. Hallintayksikkö ei yksinään riitä järkevään ohjelma-lisäkkeiden hallintaan, sillä ohjelmalisäketyyppejä tulee järjestelmään useita kappaleita. Ohjelmalisäkkeille on rakennettava omat palvelinyksiköt tyyppikohtaisesti. [17]

Ohjelmalisäkkeiden lataaminen suoritetaan ohjelma-ajon aikana. Tämä mahdollistaa turhien ohjelmalisäkkeiden sulkemisen tai uusien lataamisen kesken ohjelma-ajon. Ajon aikainen lataaminen mahdollistaa myös tarkistusmenetelmän, joka osaa seurata ohjelmalisäkkeiden latauksen onnistumista eikä tällöin pysäytä koko ohjelmaa mikäli lisäke ei löydy järjestelmästä. [18]

3.5 Rajapintaohjelmointi

Järjestelmä käyttää dynaamisia kirjastoja, joita käytettäessä on hyvä käyttää rajapintaohjelmointia. Rajapintaohjelmointi antaa mahdollisuuden käyttää luokan määritystä ilman tietoa sen toteutuksesta. [19]

Pääohjelman ei tarvitse tietää minkälainen toteutus rajapinnalla iTartarus3D on. Kuva 10 havainnollistaa kuinka rajapintatoteutuksen avulla staattinen kirjasto voi yhdistää iTartarus3D:n toteutuksen joko #1 dynaamiseen kirjastoon tai #2 dynaamiseen kirjastoon. Ohjelma voi käyttää rajapinnan avulla näin useita eri järjestelmiä ilman että esimerkiksi funktiokutsuja pitäisi muokata.



Kuva 10. Rajapintaohjelmoinnin käyttöesimerkki.

Tartarus3D:n rajapintaohjelmointi ei tule tarvitsemaan staattisen kirjaston linkittämistä useampaan dynaamiseen kirjastoon. Tartarus3D:hen rakennetaan yksi pääjärjestelmä, joka puolestaan voi ohjelmalisäkearkkitehtuurin avulla hallita useampia dynaamisia kirjastoja.

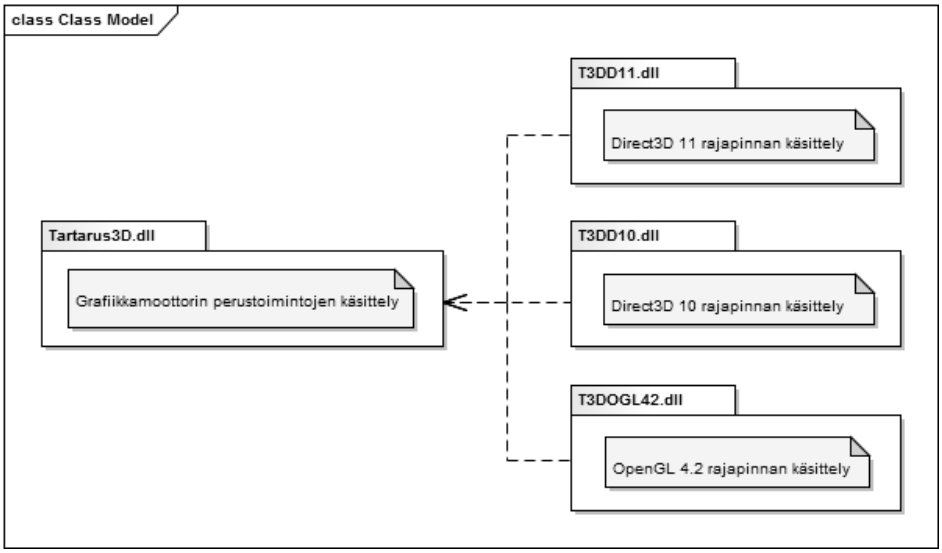
Vaihtoehtoinen menetelmä rajapintaohjelmoinnille olisi käyttää suoria funktio-osoituksia dynaamisten kirjastojen funktioihin. Suorilla funktio-osoituksilla olisi mahdollista välttää dynaamisten kirjastojen ja rajapintaohjelmoinnin luoma virtuaalifunktiokartoituksen pieni ylimääräinen resurssimenetys. Funktio-

osoittimien hallinta on kuitenkin hyvin epäkäytännöllistä kun osoittimien lukumäärä kasvaa hyvin suureksi. Tällöin olisi mahdollista että funktio-osoittimet aiheuttavat jossain määrin osoituksia myös vääriin muistialueisiin, varsinkin siinä vaiheessa, kun grafiikkamoottorista vapautetaan ohjelmalisäke kesken ohjelma-ajon. Rajapintaohjelmointi toimii varmemmin ja tarjoaa vaivattomamman pääsyn dynaamisten kirjastojen tarjoamiin luokkiin.

3.6 Kuvanmuodostuslohko

Kuvanmuodostuslohko sisältää grafiikkaohjelmointirajapintakohtaiset toiminnot. Grafiikka-moottori on mahdollista rakentaa vain yhden grafiikka-ohjelmointirajapinnan toiminnan kanssa Horde3D:n tavoin, mutta rajapinnan kovakoodaaminen järjestelmään ei kuitenkaan ole joustava ratkaisu, eikä näin ollen vastaa vaatimuksia joita Tartarus3D:lle on asetettu.

Tartarus3D:lle valittu arkkitehtuuripohja mahdollistaa kuvanmuodostus-järjestelmien tuonnin grafiikkamoottorille ohjelmalisäkkeinä kuvan 11 havainnollistamalla tavalla. Tekniikka mahdollistaa Tartarus3D:n vastaanottamaan myös tulevaisuudessa ilmestyvät grafiikkaohjelmointirajapinnat vain rakentamalla uudelle rajapinnalle oma ohjelmalisäke.

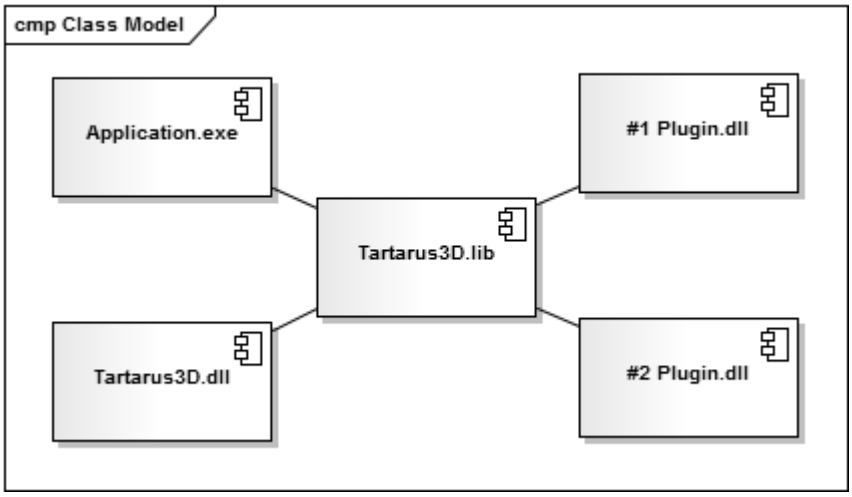


Kuva 11. Kuvanmuodostuslohkot eriytetään omiin dynaamisiin kirjastoihin.

3.7 Apukirjasto

Ohjelmalisäkearkkitehtuuri jakaa grafiikkamoottorin toiminnan useampaan ulkoiseen järjestelmään, joten yhteiset luokat kerätään erilliseen apukirjastoon. Yhteisen apukirjaston avulla on mahdollista välttää saman ohjelmakoodin kirjoittaminen useaan otteeseen. Apukirjasto toteutetaan staattisena kirjastona, koska se on helppo sisällyttää tarvittaessa kaikkialle, missä sitä tarvitaan. [20]

Apukirjastoon lisätään alustavasti geometrisessa matematiikassa tarvittavia objekteja sekä tietorakenteita, joita on kirjaston avulla mahdollista hyödyntää useassa kohteessa kuten kuvassa 12. Kirjastoon on myöhemmin helppo lisätä lisää luokkia tai funktioita, joita mahdollisesti voidaan tarvita useassa kohteessa.



Kuva 12. Apukirjastoa voidaan hyödyntää useassa järjestelmässä.

Apukirjaston luonnin sijasta olisi mahdollista myös rakentaa jokaiseen dynaamiseen kirjastoon omat räätälöidyt versiot tietorakenteista ja matematiikkaobjekteista. Tällöin tietorakenteiden päivitys ja optimointi eivät olisi yhtä vaivattomia, mutta se selkeyttäisi ohjelman toimintaa poistamalla ohjelman sidonnaisuus yhteiseen staattiseen kirjastoon. Tartarus3D:hen staattinen kirjasto kuitenkin rakennetaan, sillä sen käyttämät objektit ovat toiminnallisuuksiltaan samanlaiset jokaisessa lohkossa ja ohjelmalisäkkeessä.

4 TOTEUTUS

4.1 Käytettävät teknologiat ja työkalut

Toteutuksen ohjelmointikielenä käytetään C++-ohjelmoinkieltä. Se tarjoaa järjestelmälle asetetut muistinhallintaan liittyvät vaatimukset suorilla muistiosoittimilla. Lisäksi kielestä löytyy luonnollinen tuki olio-ohjelmoinnille. Kohdepeliarkkitehtuuri on myös rakennettu C++-kielellä, jolloin pyritään myös välttämään yhteensopivuudesta mahdollisesti aiheutuvia ongelmia.

Toinen mahdollinen ohjelmointikielivaihtoehto muistinkäsittelyn ja dynaamisten linkkikirjastojen tuen puolesta olisi kirjoittaa toteutus käyttämällä C-kieltä. C-kieli ei sisällä olio-ohjelmoinnin tukea, jolloin sen käyttäminen arkkitehtuurissa ei kuitenkaan vastaisi järjestelmälle asetettuja vaatimuksia.

Ohjelmistokehitysympäristönä käytetään Microsoft Visual Studio 2010 -ohjelmaa. Visual Studion avulla on mahdollista toteuttaa ja hallinnoida helposti useammasta järjestelmästä koostuva kokonaisuus.

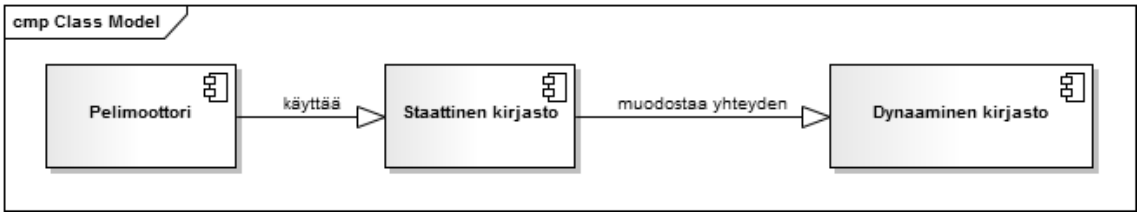
Järjestelmän lokitiedostojen seurantaan hyödynnetään ohjelmaa nimeltä Tail for Win32. Tail for Win32 on vapaan lähdekoodin ohjelma, jonka avulla on mahdollista seurata tiedostoon tapahtuvia muutoksia reaaliajassa [21].

4.2 Pääohjelman ja grafiikkamoottorin välinen yhteys

Tartarus3D:n pohjarakenne rakennetaan dynaamisena linkkikirjastona, ja sitä tullaan käyttämään päämoottorissa muodostamalla siihen ohjelma-ajon aikainen yhteys. Latausmenetelmä toteutetaan rakentamalla staattinen apukirjasto. Apukirjasto sisältää yhteyden avaukseen sekä sulkemiseen tarvittavat funktiot, ja se toimii myös varastona rajapintaluokille.

Kuvan 13 mukainen silta pelimoottorin ja dynaamisen kirjaston välillä mahdollistaa grafiikkamoottorin käytön suoraan pelimoottorista. Staattinen

kirjasto huolehtii rajapintojen latauksesta ja pitää samalla huolen siitä, että tarvittavat toteutukset ovat saatavilla dynaamisessa kirjastossa.



Kuva 13. Staattinen kirjasto muodostaa yhteyden dynaamiseen kirjastoon.

Staattinen kirjasto muodostaa yhteyden dynaamiseen kirjastoon seuraavan menetelmän mukaan:

- Ladataan dynaaminen kirjasto ja tallenna dynaamisen linkkikirjaston kahva.
- Paikannetaan kirjastosta luonti- ja sulkemisfunktiot, sekä tallenna niiden funktio-osoittimet.
- Kutsutaan luontifunktiota valmistamaan instanssi moottorista.

Yhteyden sulkeminen tapahtuu käytännössä päinvastaisella menetelmällä:

- Kutsutaan sulkemisfunktiota.
- Suljetaan yhteys käyttämällä dynaamisen linkkikirjaston kahvaa.

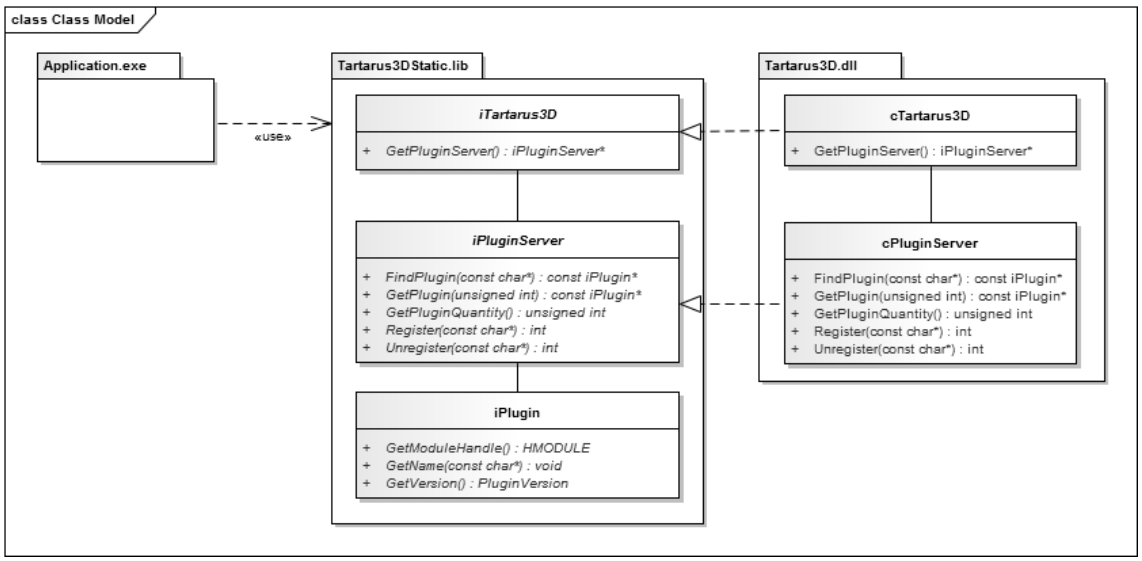
Vaihtoehtoisesti staattisen kirjaston sijasta olisi mahdollista rakentaa pelimoottoriin menetelmä, jonka avulla olisi mahdollista hakea dynaamisen kirjaston tarjoama toiminnallisuus funktio-osoittimien avulla. Menetelmä kuitenkin aiheuttaisi pelimoottoriin tilanteen, jossa sen tulisi pitää kirjaa hyvin suuresta määrästä funktio-osoittimia. Funktio-osoittimet voisivat helposti aiheuttaa myös tilanteen, jossa osoitin viittaa väärään muistialueeseen, jolloin ohjelma-ajo keskeytyisi. Dynaaminen kirjasto tulee kuitenkin sisältämään hyvin suuren määrän erilaisia funktioita, joita pelimoottori tarvitsee grafiikan muodostukseen ja näin ollen on varmempaa ja tehokkaampaa käyttää erillistä staatista kirjastoa siltana pelimoottorin ja dynaamisen kirjaston välillä.

4.3 Ohjelmalisäkejärjestelmän toteuttaminen

4.3.1 Ohjelmalisäkkeiden lisääminen ja poistaminen

Ohjelmalisäkkeiden rekisteröinnin tai rekisteristä poistavan komennon tulee olla helposti käytettävissä. Yksi vaihtoehto on rakentaa aina uutta toiminnallisuutta tarvittaessa uusi funktiokutsu iTartarus3D-rajapintaan. Kuitenkin pidemmälle ajateltuna siitä saattaa aiheutua tulevaisuudessa haittaa, sillä iTartarus3D-rajapintaan alkaa melko varmasti kertyä hyvin paljon erilaisia funktioita. Parempi vaihtoehto on muodostaa rajapinnasta linkki iPluginServer-rajapintaan, jonka avulla voidaan listata funktiot, joita on tarkoitettu kutsuttavaksi myös moottorin ulkopuolelta.

Ohjelmalisäkkeen lataaminen järjestelmään suoritetaan kutsumalla ohjelmalisäkkeiden hallinnasta vastaavaa cPluginServer-luokkaa. Kuva 14 havainnollistaa, miten cPluginServer periytetään rajapinnasta iPluginServer, jolloin Application.exe:stä on mahdollista suorittaa cPluginServerin funktioita.



Kuva 14. Ohjelmalisäkejärjestelmän luokkakaavio.

Hallintayksiköiden tehtävä on tarjota erilaisia funktioita niiden hallitsemien objektien yleiseen käsittelyyn. Alustavasti ohjelmalisäkepalvelimelle luodaan seuraavanalaiset funktiot perustoimintoja varten:

- Register rekisteröi uuden ohjelmalisäkkeen. Parametrina funktio tarvitsee ohjelmalisäkkeen tiedostonimen.
- Unregister toimii täysin päinvastoin kuin register-funktio. Parametrina funktio tarvitsee ohjelmalisäkkeen tiedostonimen.
- FindPluginin avulla on mahdollista hakea ohjelmalisäke nimen mukaan.
- GetPluginin avulla on mahdollista hakea ohjelmalisäke sen listaindeksin mukaan.
- GetPluginQuantityn avulla on mahdollista hakea tieto siitä, montako ohjelmalisäkettä järjestelmään on ladattuna.

Tuhoutumisvaiheessa cPluginServer-luokka tuhoaa osoittimet kaikkiin ohjelmalisäkkeisiin ja vapauttaa niille kuuluvat kahvat. Rakennusvaiheessa cPluginServer tarvitsee cTartarus3D:n osoittimen, jonka se voi antaa eteenpäin ohjelmalisäkkeiden rekisteröintivaiheessa.

4.3.2 Ohjelmalisäkkeiden hallinta

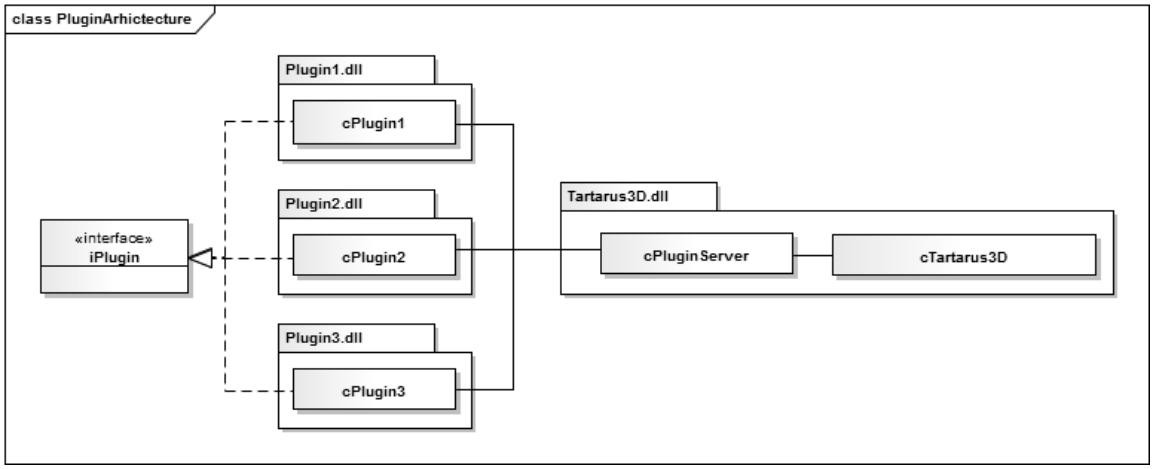
Ohjelmalisäkkeet rekisteröidään aluksi aina ohjelmalisäkerekisteriin. Ohjelmalisäkerekisteri pitää ohjelmalisäkkeen tiedot tallessa niin että niitä pääsee tarvittaessa käsittelemään. Eräitä tärkeitä piirteitä ohjelmalisäkkeen tallentamisessa rekisteriin ovat ainakin seuraavat:

- Ohjelmalisäkkeen nimeä voidaan myös käyttää ohjelmalisäkkeen tunnistukseen. Tartarus3D:ssä tämä on sama kuin tiedostonimi.
- Funktio-osoitin osoittaa ohjelmalisäkkeen rekisteröintifunktioon.
- Funktio-osoitin osoittaa ohjelmalisäkkeen rekisteristä poistamisesta huolehtivaan funktioon.
- Rajapinta-osoitin osoittaa ohjelmalisäkkeen luokkaan (kaikki ohjelmalisäkkeet periytyvät rajapinnasta iPlugin)

- Ohjelmalisäkkeen kahvan avulla on mahdollista vapauttaa dynaaminen linkkikirjasto kun ohjelmalisäkettä ei enää tarvita.

Rekisteröityneiden ohjelmalisäkkeiden hallinta toteutetaan ohjelmalisäke-palvelimen cPluginServerin kautta. Sen avulla on mahdollista hallita järjestelmän ohjelmalisäkkeitä esimerkiksi lataamalla tai vapauttamalla niitä kesken ohjelma-ajon.

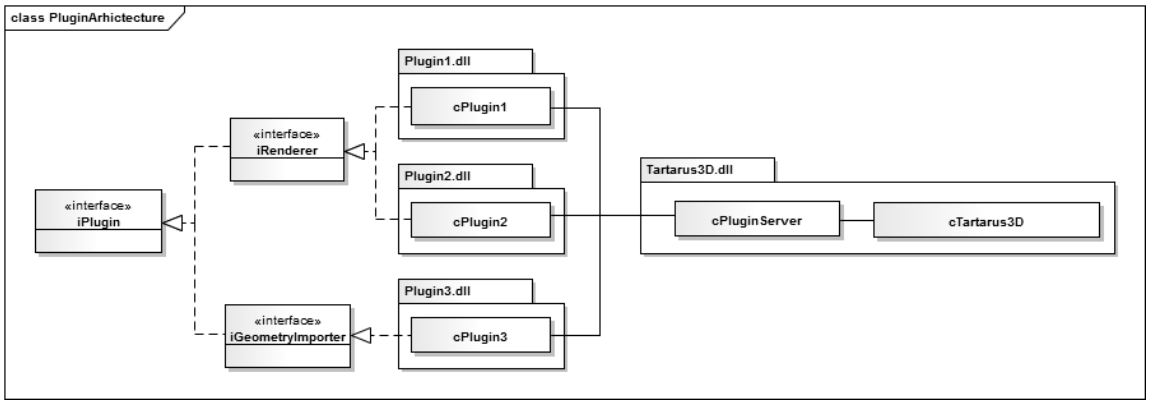
Kuvan 15 mukainen periyttäminen vain yhdestä rajapinnasta antaa mahdollisuuden tallentaa ohjelmalisäkkeiden osoittimet vain yhteen tietorakenteeseen. Arkkitehtuurin kanssa ohjelmalisäkkeiden käytössä olevat funktiot rajoittuvat iPlugin-rajapinnan sisältämiin funktioihin, joka ei ole toivottava ratkaisu. Ohjelmalisäkkeitä ei näin ollen tule periyttää pelkästään iPlugin-rajapinnasta, koska tällöin iPlugin-rajapinnan pitäisi listata kaikki mahdolliset ohjelmalisäkkeiden hallintaan tarvittavat funktiot.



Kuva 15. Ohjelmalisäkkeet periytyvät iPlugin-rajapinnasta.

Arkkitehtuuria tulee kehittää paremmaksi rakentamalla iPlugin-rajapinnan ja toteutusten väliin vielä oma ohjelmalisäkekohtainen rajapinta, joka listaa ohjelmalisäketyypille olennaiset funktiot. Tällä tavalla tyyppikohtaisten ohjelmalisäkepalvelinten on mahdollista käyttää oikein ohjelmalisäkkeiden tarjoamia funktiota, eikä ole tarvetta tehdä kaikkia mahdollisia funktiototeutuksia kaikille ohjelmalisäkkeille.

Kun ohjelmalisäkkeet periytetään useamman rajapinnan kautta, on mahdollista jakaa ohjelmalisäkkeille yhteiseksi valitut ominaisuudet iPlugin-rajapinnan avulla ja ohjelmalisäketyyppikohtaiset funktiot puolestaan välirajapinnan avulla. Kuva 16 havainnollistaa, miten välirajapintoina toimivat alustavasti iRenderer- ja iGeometryImporter-rajapinnat.



Kuva 16. Paranneltu menetelmä ohjelmalisäkkeiden periyttämiselle.

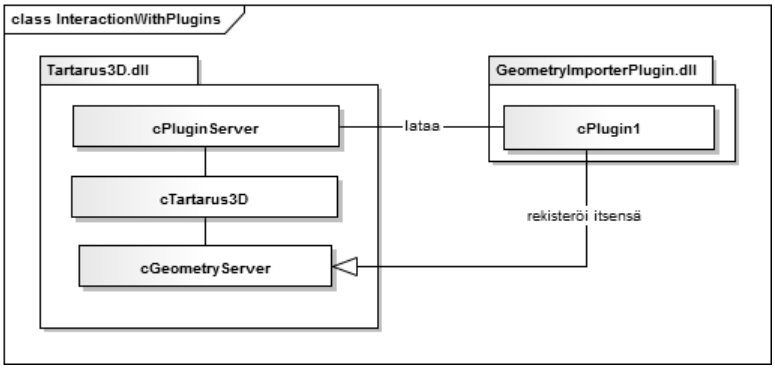
Ohjelmalisäkejärjestelmästä vastaava cPluginServer tulee sulkea vasta Tartarus3D:n pääjärjestelmän sulkemisvaiheessa. Mikäli ohjelmalisäkepalvelin suljetaan liian aikaisessa vaiheessa, saattaa se aiheuttaa järjestelmään tietovirran katkoksen kriittisessä kohdassa. Tartarus3D:n sammutusvaiheessa ohjelmalisäkejärjestelmä tullaan siis aina sulkemaan aivan viimeisenä.

4.3.3 Ohjelmalisäkepalvelimet

Tartarus3D:hen rakennetaan ohjelmalisäketyyppien mukaisia ohjelmalisäke-palvelimia, joiden tehtävä on seurata tyyppikohtaisesti millaisia ohjelmalisäkkeitä on rekisteröitynyt järjestelmään. Ohjelmalisäkkeet rakennetaan menetelmän mukaan, jossa ohjelmalisäke hoitaa itse rekisteröitymisen sen toiminnan kannalta oleelliseen palvelimeen.

Kuva 17 havainnollistaa miten ohjelmalisäkkeiden rekisteröintimenetelmässä pääohjelmalisäkepalvelin cPluginServer antaa rekisteröintifunktiolle parametriksi pääjärjestelmän osoittimen, jolla ohjelmalisäke pääsee käsiksi

kaikkiin järjestelmän tyyppikohtaisiin ohjelmalisäkepalvelimiin. Ohjelmalisäke kykenee rekisteröimään tällöin itsensä valitsemaansa palvelimeen kutsumalla tyyppikohtaisen palvelinluokan rekisteröintifunktioita. Tällöin ei ole myöskään tarvetta rakentaa pääohjelmalisäkepalvelimeen minkäänlaisia tarkistus-menetelmiä tunnistamaan ohjelmalisäkkeen tyyppiä. Tyyppikohtaisen ohjelmalisäkepalvelimen ei tarvitse tehdä muuta kuin huolehtia ohjelmalisäkkeiden lisäämisestä ja vapauttamisesta.



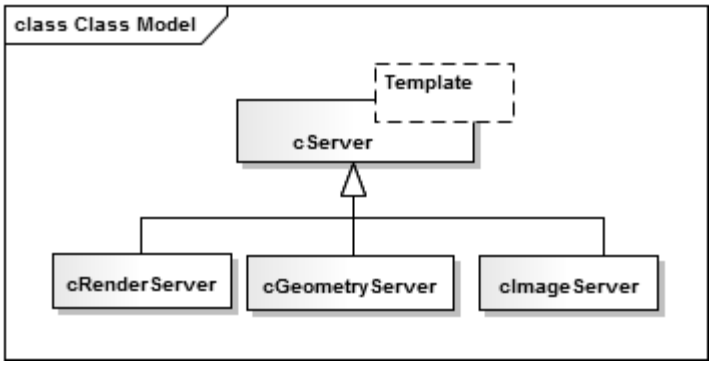
Kuva 17. Geometria-ohjelmalisäke rekisteröi itsensä geometriapalvelimeen.

Ohjelmalisäkkeiden tulee olla tietoisia omasta kahvastaan. Ilman tietoa kahvasta niiden ei ole mahdollista luoda järjestelmäkohtaisia kutsuja esimerkiksi dialogien luomiseksi. Tästä johtuen ohjelmalisäkkeen latauksen yhteydessä tulee cPluginServer-luokan lähettää ohjelmalisäkeelle tieto kahvasta, joka on saatu latauksen yhteydessä.

Ohjelmalisäkkeet tarkistavat myös moottoriversion ennen rekisteröitymistään palvelimeen. Mikäli grafiikkamoottorin ohjelmaversio poikkeaa ohjelmalisäkkeen tukemasta ohjelmaversiosta, ei ohjelmalisäke rekisteröi itseään grafiikkamoottoriin.

Tyyppikohtaiset ohjelmalisäkepalvelimet sisältävät hyvin samantapaisia funktioita, joten ohjelmakoodin toistamista pyritään välttämään kirjoittamalla ohjelmalisäkepalvelinluokille malliluokka. Malliluokka on geneeristä ohjelmointia, jolloin voidaan käyttää samaa toiminnallisuutta ja pohjaa täysin eri luokkien kanssa [22].

Kuvan 18 mukainen abstrakti malliluokka sisältää tiedon ohjelmalisäkemäärästä sekä ohjelmalisäkkeiden yhteisten tietojen käsittelyyn tarvittavat funktiot. Malliluokasta periytyvät palvelinluokat sisältävät funktiot, joita tarvitaan tyyppikohtaiseen ohjelmalisäkkeiden hallintaan.



Kuva 18. Ohjelmalisäkepalvelimet periytetään cServer-malliluokasta.

Ohjelmalisäkepalvelimet rakennetaan cTartarus3D-luokassa, koska se toimii myös porttina ohjelmalisäkkeiden rekisteröinnille. Rekisteröinti tapahtuu kutsumalla cServer-luokan Register-funktiota, joka lisää ohjelmalisäkkeen osoittimen talteen ohjelmalisäkepalvelimelle.

Tartarus3D:hen toteutettu ohjelmalisäkepalvelinten geneerinen rakenne mahdollistaa aivan uudentyyppisten ohjelmalisäketyyppien liittämisen moottoriin helposti.

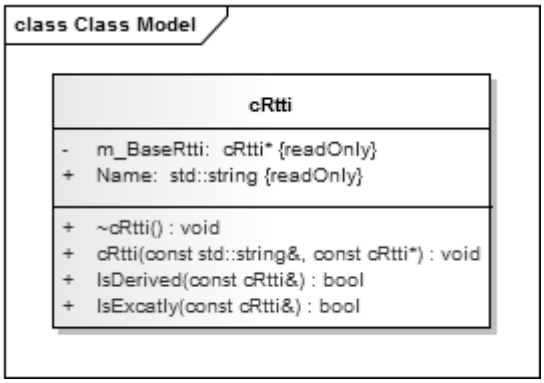
4.4 Objektijärjestelmä

Objektijärjestelmä on järjestelmässä käytettyjen objektien taustalla toimiva järjestelmä, jonka avulla on mahdollista käsitellä järjestelmän sisällä luotavia objekteja objektitason komennoilla. Kaikki Tartarus3D:n objektit periytyvät cObject-luokasta.

Objektijärjestelmän kautta on mahdollista jakaa kaikille järjestelmän objekteille niille ominaiset attribuutit ja funktiot. Tartarus3D:n objekteille luodaan oma id-numerointijärjestelmä, jonka avulla jokainen luotu objekti-instanssi on

mahdollista tunnistaa uniikin kokonaisnumeron avulla. Objektijärjestelmään rakennetaan myös laajennettu versio rtti:stä. Laajennettu versio tulee sisältämään tiedon objektin tyypistä sekä tiedon, mistä luokasta objekti on periytetty. [23]

Kuvassa 19 on esitetty kehittyneempi versio rtti:stä, joka mahdollistaa tarkistamaan, mikäli objekti periytynyt jostain tietystä luokasta. Kaikki cRtti-luokat sisältävät osoittimen yläluokan cRtti-luokkaan sekä julkisen nimen. Attribuutit ovat vakio-tyyppisiä, joten niiden määritys voidaan tehdä vain rakennusfunktion alustusluettelossa.

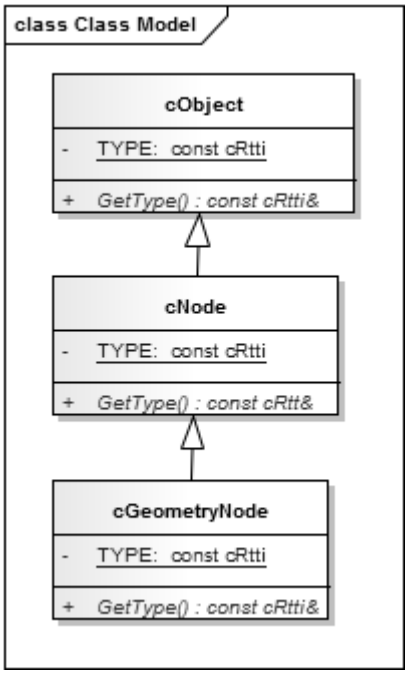


Kuva 19. Rtti-laajennuksesta vastaava luokka.

Luokka tarjoaa myös kaksi yksinkertaista funktiota avustamaan cRtti-objektien vertailua. IsExcatly-funktion avulla on mahdollista tarkistaa cRtti-luokkien samanlaisuus. IsDerived-funktion avulla on mahdollista tarkistaa cRtti-luokan periytyvyys jostain tietystä cRtti-luokasta.

Kaikkien järjestelmän objektiluokat sisältävät staattisen julkisen cRtti-attribuutin, sekä julkisen GetType-funktion. GetType-funktion tulee olla abstrakti, jotta objektista periytyvät alaluokat voivat korvata sen toiminnan.

Kuvan 20 havainnollistaa miten luokasta cObject periytyvien luokkien attribuutti TYPE on kaikissa luokissa staattinen, joka mahdollistaa luokan rtti-tyypin haun suoraan luokasta. Staattinen cRtti-attribuutti rakennetaan luokan toteutuksessa, ja sille annetaan parametreiksi luokan nimi sekä yläluokka.



Kuva 20. Rtti-laajennuksen toiminta objektijärjestelmässä.

Luokalla cObject ei ole lainkaan yläluokkaa, joten sen yläluokka määritetään tyhjäksi. GetType-funktion avulla on mahdollista hakea objektin aito tyyppi, myös siinä vaiheessa kun objektiin viitataan abstraktin luokan osoittimella.

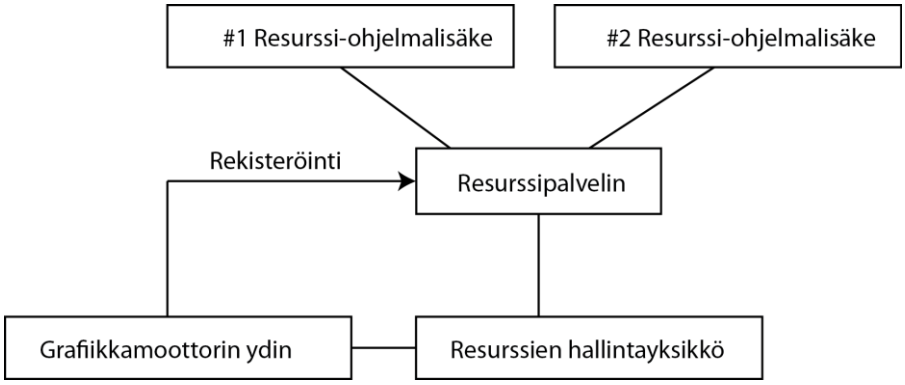
Lisäksi objektijärjestelmän pääobjektiluokalle rakennetaan avustusfunktioita, joita on mahdollista hyödyntää vertailemaan objekteja ja objektien rtti-arvoja keskenään.

4.5 Resurssijärjestelmä

4.5.1 Resurssien tuonti järjestelmään

Tartarus3D:n resurssijärjestelmän toiminta tulee perustumaan keskusvaraston toimintaan. Keskusvarastossa säilytetään resursseja, kunnes ne hävitetään manuaalisesti tai grafiikkamoottori suljetaan. Keskusvaraston avulla on mahdollista hoitaa resursseille yhteisiä toimenpiteitä, sekä laskea moottoriin tuotujen resurssien kokonaismäärä.

Grafiikkamoottorin ytimenä toimiva cTartarus3D-luokka käyttää kuvan 21 mukaisesti suoraa yhteyttä resurssipalvelimeen ainoastaan alustuksen, tuhoamisen ja ohjelmalisäkkeiden rekisteröinnin aikana. Resurssien hallinta tehdään resurssien hallintayksikössä, jonka avulla suoritetaan resurssien latausoperaatiot.



Kuva 21. Resurssijärjestelmän yleiskuvaus.

Resurssipalvelin ja resurssien hallintayksikkö on eriytetty toisistaan, koska resurssien hallintayksikköä halutaan käyttää vapaasti eri puolella järjestelmää. Ohjelmalisäkejärjestelmä ei salli singleton-suunnittelumallin staattisten objektien käyttöä ohjelmalisäkkeiden rekisteröinnissä, joten varsinainen resurssipalvelin rakennetaan, kuten muutkin ohjelmalisäkepalvelimet. Resurssipalvelimen hallinta suoritetaan erillisen hallintayksikön kautta, jolloin ulkopuoliset ohjelmalisäkkeet on mahdollista yhdistää järjestelmään suunnitellun menetelmän mukaan.

Resurssipalvelin tarkistaa, mitä resurssiohjelmalisäkettä sen tulee hyödyntää resurssin luontiin käyttämällä tiedostonimen päätettä. Oikean resurssiohjelmalisäkkeen löytyessä järjestelmä tarkistaa minkä tyyppinen resurssi on. Resurssin tyyppitieto pystytään hakemaan resurssi-ohjelmalisäkeestä.

Kun resurssipalvelin on saanut resurssityypin määrityksen ja varmistanut, että siihen löytyy sopiva ohjelmalisäke, se luo uuden resurssin ja antaa siitä

osoittimen iResource-tyyppisenä resurssi-ohjelmalisäkkeelle. Resurssi-ohjelmalisäke rakentaa uuden objektin ja antaa tiedon onnistuiko operaatio.

Vaihtoehtoinen menetelmä olisi rakentaa vain resurssipalvelin, jonka avulla suoritettaisiin kaikki resurssien hallintaan liittyvät operaatiot. Resursseille tulisi tällöin rakentaa oma resurssitietokanta, josta resurssit ovat käytettävissä jokaisessa järjestelmän osassa. Resurssitietokanta olisi myös mahdollista toteuttaa rakentamalla resursseille yhteinen resurssiobjekti, josta resurssit periytyvät. Yhteinen resurssiobjekti voisi pitää kirjaa kaikista siitä periytyvistä luokista staattisten attribuuttien avulla. Toiminta olisi samantapainen kuin järjestelmään rakennettava yleinen objektijärjestelmä. Yhteistä resurssiobjektia olisi mahdollista käyttää hakumenetelmien avulla, jolloin jokin tietty resurssi olisi mahdollista löytää käymällä läpi yhteisestä resurssiobjektista periytetyt objektit. Menetelmän hyvänä puolena olisi se että resurssiobjektit olisi automaattisesti lisättyinä tietokantaan, jolloin ei lisäämiseen tarvittaisi omaa hallintayksikköä. Huonona puolena järjestelmästä tiettyjen resurssien poistaminen ja selaaminen olisi hidasta, sillä yhteinen resurssiyksikkö joutuisi suorittamaan suuren määrän hakuoperaatioita aina resurssien käsittelyn yhteydessä. Tartarus3D:hen rakennettava yhdistetty hallintayksikkö mahdollistaa suurempien resurssimäärien poistamisen ja hallinnoinnin huomattavasti vaivattomammin.

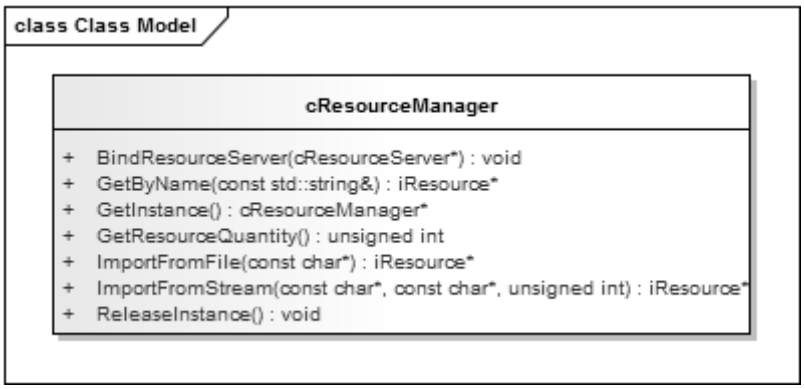
4.5.2 Hallintayksikkö

Resurssien hallinnasta vastaa cResourceManager-luokka, joka on rakennettu singleton-suunnittelumallin mukaan. Singleton-suunnittelumalli mahdollistaa saman instanssin tavoitettavuuden kaikkialla grafiikkamoottorin sisällä ja varmistaa samalla että luokasta on vain yksi instanssi olemassa [24]. Hallintayksikön avulla voidaan tuoda ulkopuolisesta lähteestä resursseja järjestelmään.

Kun hallintayksikköä käskytetään lataamaan uusi resurssi järjestelmään, tarkistaa se ensin resurssin saatavuuden järjestelmästä. Resurssit ovat aina nimikohtaisia, eli järjestelmästä ei voi löytyä kerralla esimerkiksi kahta

"objekti.obj"-resurssia. Hallintayksikkö palauttaa osoittimen saman nimen omistavaan resurssiin, mikäli resurssi on luontivaiheessa jo saatavilla.

Kuvassa 22 esitetyt funktiot kattavat resurssien luonnin ja etsinnän järjestelmästä. Järjestelmään olisi mahdollista myös luoda tiedon ulkopuoliseen tallentamiseen tarkoitetut funktiot, joiden avulla grafiikkamoottorin tila olisi mahdollista tallentaa tietokoneen tiedostojärjestelmään. Alustavasti tallentamisoperaatioiden funktiota ei toteuteta järjestelmään, koska se ei ole suunniteltu ominaisuus järjestelmässä. Tallentamisoperaatiot on mahdollista lisätä myöhemmin suhteellisen vaivattomasti. Lisäys tulee tällöin tehdä myös järjestelmän niihin ohjelmalisäkkeisiin, joihin se halutaan ottaa käyttöön.



Kuva 22. Resurssienhallinnasta vastaavan cResourceManagerin funktiot.

Resurssijärjestelmä rakennetaan kykeneväiseksi lataamaan resursseja suoraan tietovirrasta. Pelkällä tietovirralla ei ole tiedostonimeä, joten varsinaista tiedostonimeä ei voida hyödyntää resurssien saatavuuden tarkistuksessa. Ratkaisuksi otetaan käyttöön menetelmä, jossa myös tietovirrasta tuoduille objekteille pitää määrittää jonkinlainen nimi.

Menetelmän avulla on mahdollista ladata sama objekti uudestaan, mikäli käyttäjä nimeää sen eri tavalla. Tartarus3D:n kohdalla tietovirrat syötetään grafiikkamoottorille IOMS-järjestelmän kautta, jonka avulla toteutetaan myös tietovirtojen nimeäminen.

Luokan cResourceManager sulkeminen toteutuu kutsumalla luokan ReleaseInstance-funktiota. Sulkemisen yhteydessä tuhotaan resurssivarasto,

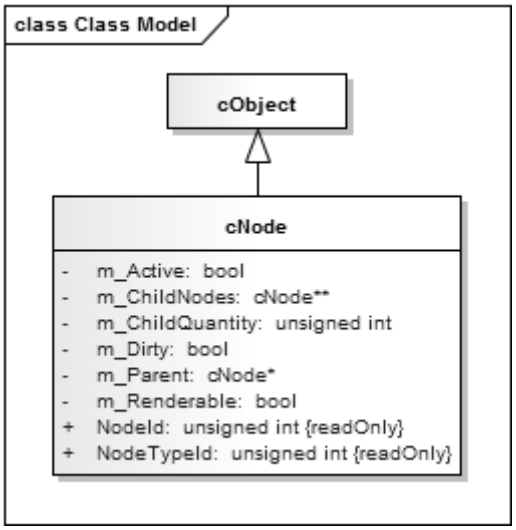
jonka avulla vapautetaan myös kaikki resursseille varattu muisti. ReleaseInstance-funktio tulee ajaa grafiikkamoottorin sulkuvaiheessa, koska muutoin objektit jotka käyttävät resursseja voivat tehdä järjestelmään virheitä.

4.6 Näkymägraafi

4.6.1 Solmujen rakenne

Tartarus3D:n graafin perusideana on, että solmulla voi olla yksi vanhempi ja useita lapsia. Tartarus3D:n graafin solmut periytetään luokasta cNode. Luokka sisältää solmuille yleisiä operaatioita, sekä solmujen keskinäiseen vertailuun tarkoitettuja apufunktioita. Luokka on abstrakti, joten siitä ei pysty suoraan luomaan instanssia.

Kuvassa 23 esitetyn esimerkin mukaan kaikkien solmujen tulee vähintään tietää lapsisolmujensa määrä sekä osoitin vanhempaansa. Solmun kaaret ovat tällä tavoin kaksisuuntaisia, jolloin on mahdollista hakea tietoa joko solmun lapsilta tai vanhemmalta.



Kuva 23. cNode-luokan attribuutit.

Solmuilla on oma solmukohtainen tunniste. NodeId-attribuutti on jokaiselle solmulle uniikki. Uniikin tunnisteen ansiosta solmu voidaan aina erottaa muista

solmuista, joten siihen on mahdollista aina viitata suoraan. NodeTypeId on solmun tyyppi, joka voi olla sama usealla solmulla. Solmun tyyppiä voidaan hyödyntää yhdessä rtti-järjestelmän kanssa.

Solmuluokalle luodaan attribuutit aktiivisuustilan, likaisuuden ja kuvan-muodostuksen tilan näyttämiseen tarkoitetut attribuutit. Aktiivisuustila on yleinen tila, joka määrittää, onko solmu käytössä. Jos aktiivisuus on poissa käytössä, ei myöskään solmun lapset ole käytössä. Likaisuustilalla kerrotaan, onko solmulla tarvetta päivitykselle. Päivitys tehdään, kun solmuja läpikäydään graafin läpikäyntivaiheessa. Kuvanmuodostusattribuutti kertoo, onko solmun sisältö mahdollista muodostaa kuvaksi.

Luokalla cNode on funktiota solmujen haku-, liitos- ja tieto-operaatioille. Tärkeimmät toiminnot ovat lapsisolmujen haku sekä abstraktit funktiot päivityksestä ja kuvanmuodostuksesta.

4.6.2 Solmujen luonti

Näkymägraafin solmutyyppien määrä halutaan pitää dynaamisena tulevaisuuden kehityksen kannalta. Tartarus3D:hen kehitettään dynaaminen luontiyksikkö, koska solmuja halutaan lisätä jatkossa saman luontimenetelmän pariin.

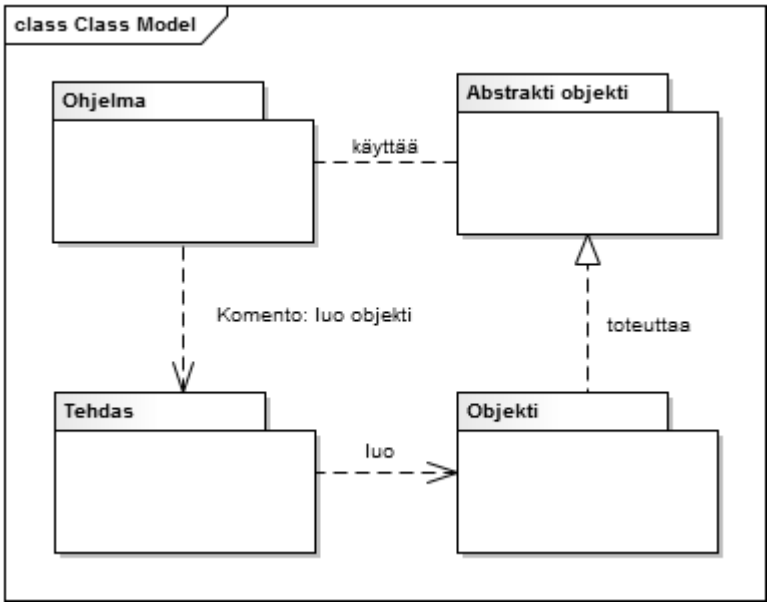
Solmujen rakentamiseen olisi mahdollista käyttää solmujen omia rakennusfunktioita, mutta solmut eivät olisi silloin mitenkään keskitettyjä. Keskittyneisyyden puutos estäisi solmujen hallinnan keskitetysti, jolloin esimerkiksi grafiikkamoottorin kaikkien solmujen tuhoaminen kerralla olisi vaivalloista.

Yksi vaihtoehto on käyttää *Factory Method* -suunnittelumallia, jolloin cNode-luokkaan rakennettaan abstrakti tehdasfunktio. Tehdasfunktion avulla objektien luonti palauttaa abstraktin luokan osoittimen, jota on mahdollista käyttää graafissa. Tehdasfunktio ei kuitenkaan ratkaise solmujen keskitysongelmaa, vaan solmujen yhteinen hallinta jäisi edelleen avoimeksi. [14]

Horde3D:ssä on kehitetty jalostettu versio solmujen luonnista. Luontimenetelmä perustuu solmujen rekisteröintiin ja luontiin keskitettyyn hallintayksikköön. Jokaisesta solmutyypistä annetaan tieto tyypistä, tyyppinimestä, sekä kolmen funktion osoittimet. Kolme funktio-osoitinta osoittaa rekisteriin lisättävän solmun tietojen hakuun, luontiin ja kuvanmuodostukseen.

Tartarus3D:n solmujen luontiin hyödynnetään Horde3D:n menetelmää yhdessä *Abstract Factory* ja *Factory Method* -suunnittelumallien kanssa. Solmujen luontiin rakennetaan tehdas cNodeFactory, johon voidaan rekisteröidä solmutyyppejä. Solmutyyppien rekisteröinnin yhteydessä tehtaalle annetaan tieto solmun tyypistä, tyyppinimestä ja tehdasfunktion osoittimesta. Jokaiselle solmutyypille rakennetaan staattinen tehdasfunktio, joka palauttaa cNode-tyyppisen osoittimen. [14, 25]

Kuva 24 havainnollistaa, miten Tartarus3D:n solmutehdas rakentaa uuden solmun, kun tehtaan luontifunktiota kutsutaan. Luontifunktiossa tehdas tarkistaa, minkä tyyppinen solmu halutaan rakentaa ja katsoo samalla onko solmutyyppi rekisteröity tehtaaseen. Mikäli solmutyyppi löytyy tehtaan solmurekisteristä, tehdas käyttää solmutyypin staattista tehdasfunktiota rakentaakseen solmun. Tehdas lisää rakentamisen jälkeen solmun osoittimen solmulistaan ja palauttaa osoittimen myös luontia kutsuneelle järjestelmälle.

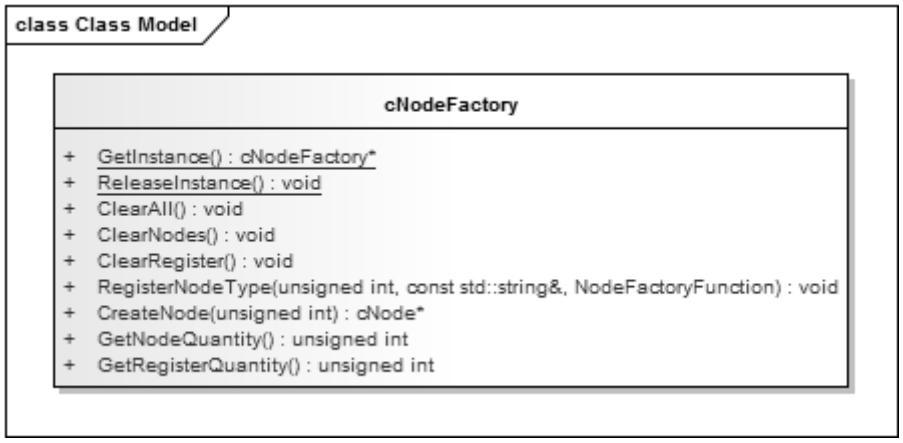


Kuva 24. Solmujen luontimenetelmän toiminta Tartarus3D:ssä.

Tartarus3D:n solmutehdas sisältää osoittimet kaikkiin järjestelmän solmuihin. Solmutehtaan avulla on mahdollista tuhota kaikki järjestelmässä käytössä olevat solmut sulkemalla tehdas tai kutsumalla tyhjennysfunktiota. Tehdas tyhjentää tuhoutuessaan myös oman solmurekisterinsä.

Solmutehdas rakennetaan singleton-suunnittelumallin mukaan, jonka avulla on mahdollista varmistaa että tehtaasta on vain yksi instanssi olemassa ohjelma-ajon aikana ja samalla se on tavoitettavissa kaikkialla. [24]

Kuvan 25 mukainen solmutehdas tarjoaa mahdollisuuden tyhjentää oman solmutyyppirekisterinsä tai solmut joita se on luonut ja operaatiot uusien solmutyyppien rekisteröinnille ja luomiselle. Tartarus3D:hen luodaan alustaviksi solmuiksi geometria-, kamera-, juuri-, näkymä-, tila- ja liikesolmut. Alustavien solmutyyppien avulla pohja-arkkitehtuurin toimintaa on mahdollista testata käytännössä.



Kuva 25. Solmutehdasluokan funktiot.

Geometriasolmun tehtävä on pitää sisällään geometriaan liittyvää tietoa, jossa voidaan myös viitata geometriaresursseihin. Kamerasolmut sisältävät linkin kameraobjektiin, jota käytetään kuvanmuodostuksen apuna. Juurisolmut luodaan tulevaisuuden varalta omaksi luokakseen tulevaisuuden varalta, mutta alustavasti niillä ei ole muuta toiminnallisuutta kuin osoittaa näkymägraafin aloituspistettä. Tilasolmut käsittelevät kuvanmuodostustiloja, kuten

pinnoitusväriä tai varjostustilaa. Liikesolmujen avulla voidaan määrittää niiden lapsisolmujen rotaatio, paikka ja kokosuhteet.

4.7 Kuvanmuodostusjärjestelmä

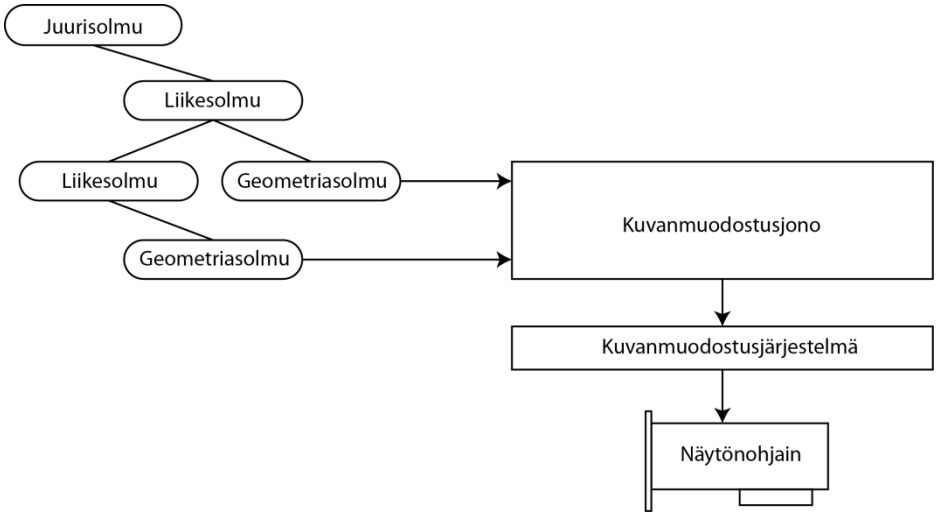
Tartarus3D:n kuvanmuodostusjärjestelmät toteutetaan rajapinnasta iRendererPlugin, joka sisältää pakolliset kuvanmuodostusjärjestelmän funktiot. Kuvanmuodostus-ohjelmalisäkkeet rekisteröityvät muiden ohjelmalisäkkeiden tapaan järjestelmään, mutta ne eivät aktivoi itseään heti.

Kuvanmuodostusjärjestelmä aktivoidaan vasta kun Tartarus3d.dll:n cRendererServer-luokka on saanut määrityksen minkä kuvanmuodostusjärjestelmä käyttäjä on valittu järjestelmän aktiiviseksi. Tartarus3D:ssa vain yksi kuvanmuodostusjärjestelmä voi olla kerrallaan aktiivinen.

4.7.1 Kuvanmuodostusjono

Tartarus3D:n kuvanmuodostusjärjestelmä rakennetaan käyttämään puskuroitua kuvanmuodostusta. Puskuroidun kuvanmuodostuksen avulla on mahdollista välttää ylimääräisiä tilavaihdoksia piirtoputkeen. Tilavaihdokset saadaan minimoitua järjestelemällä kuvanmuodostusoperaatiot efektikohtaisesti. Tartarus3D:hen rakennetaan kuvanmuodostusjono, joka toimii välipuskurina näkymägraafin ja kuvanmuodostusjärjestelmän välillä.

Kuva 26 havainnollistaa, miten kuvanmuodostusjärjestelmässä suoritettavia operaatioita kerätään näkymägraafin täyttämästä kuvanmuodostusjonosta. Kuvanmuodostusjonossa operaatiot lajitellaan efektien mukaan, jonka jälkeen ne syötetään kuvanmuodostusjärjestelmän kautta näytönohjaimelle.



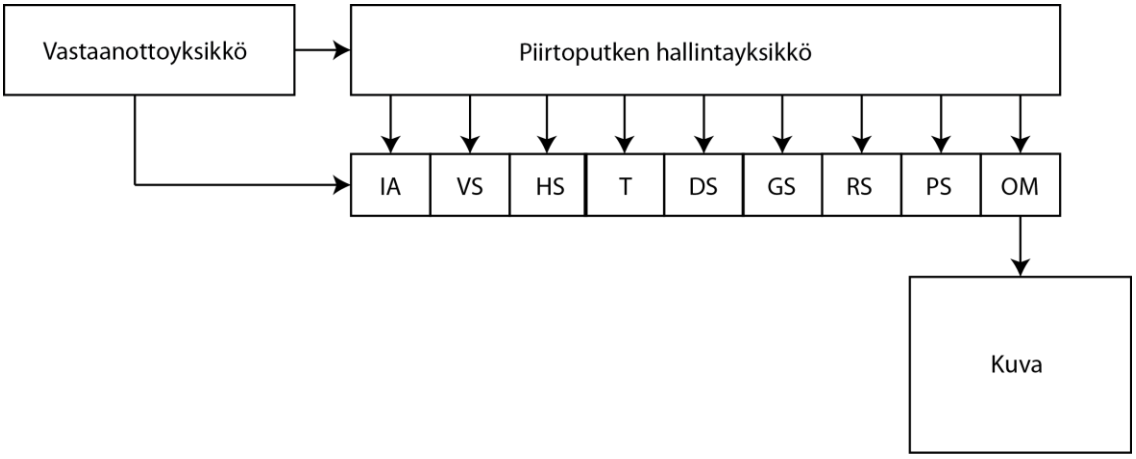
Kuva 26. Kuvanmuodostusjonon osuus järjestelmässä.

4.7.2 Piirtoputken hallintayksikkö

Kuvanmuodostuslohkon arkkitehtuuriin rakennetaan hallintajärjestelmä, jonka avulla on mahdollista vaihtaa piirtoputken tilaa. Menetelmä perustuu kuvanmuodostusjonosta saapuvan tiedon attribuuttien ja asetuksien tarkkailuun.

Piirtoputken hallintaa varten rakennetaan oma luokka, jonka käskytys voidaan suorittaa suoraan kuvanmuodostusjärjestelmän pääluokasta cT3DD11. Tilavaihdokset ovat kuvanmuodostusjärjestelmän yksi pullonkauloista, joten ne toteutetaan yksinkertaisia switch-case-rakenteita hyväksikäyttäen.

Tartarus3D:n kuvanmuodostuksen vastaanottoyksikkönä toimii cT3DD11-luokka, joka ottaa kuvanmuodostusoperaatioita Render-funktion avulla. Funktio keskustelee ensin piirtoputken hallintayksikön kanssa, jonka jälkeen se syöttää kuvanmuodostuksen tiedon piirtoputken ensimmäiselle osalle kuvan 27 mukaisella tavalla.



Kuva 27. Kuvaus T3DD11.dll:n piirtoputken hallintayksikön toiminnasta. Kuvassa esiintyvät piirtoputken osat ovat Direct3D:n piirtoputken osien lyhenteitä.

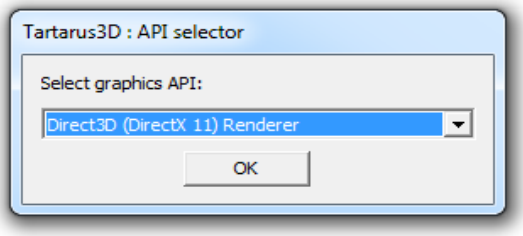
Kun tieto on läpäissyt piirtoputken, voidaan se esittää kuvanmuodostuksessa käytettävään kohdepintaan. Kohdepintana voidaan käyttää esimerkiksi peli-ikkunaa.

5 TULOKSET

Lopputuloksena syntyi ohjelmalisäkearkkitehtuuri, joka kattaa resurssien ja kuvanmuodostuksen hallinnan. Ohjelmalisäkearkkitehtuurin avulla järjestelmään on mahdollista ladata useampia dynaamisia linkkikirjastoja ja ne laajentavat järjestelmän toimivuutta. Arkkitehtuurille asetettu päävaatimus dynaamisesta pohja-arkkitehtuurista saatiin toteutettua.

Ohjelmalisäkearkkitehtuuri mahdollistaa järjestelmäkomponenttien latauksen tapauskohtaisesti. Lisäksi ohjelmalisäkkeet ovat käytettävissä myös muissa ohjelmissa, mikäli niille rakennetaan uusi rekisteröintifunktio.

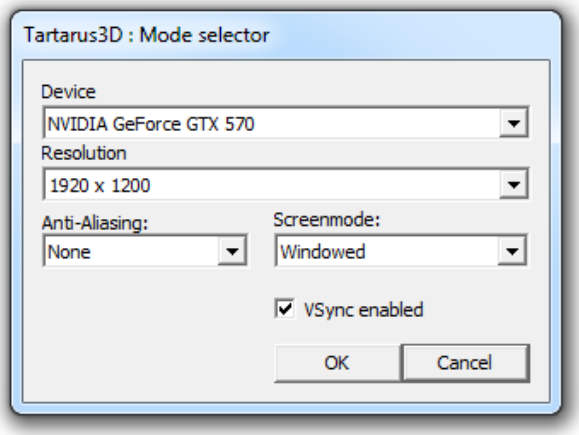
Kuvanmuodostusjärjestelmien tunnistamiseen kehitettiin yksinkertainen grafiikkaohjelmointirajapintavalitsin, jonka avulla käyttäjä voi valita moottorin käynnistyksen yhteydessä haluamansa grafiikkaohjelmointirajapinnan. Kuvassa 28 esiintyvä grafiikkaohjelmointirajapintavalitsin on tarkoitettu vain kehitys- ja testauskäyttöön, joten järjestelmään rakennettiin mahdollisuus tehdä rajapinta-valinta myös ilman dialogin käyttöä.



Kuva 28. Tartarus3D:n grafiikkaohjelmointirajapinta-valitsin.

Pohja-arkkitehtuurin ohessa kehitetty kuvanmuodostusohjelmalisäke T3DD11 toimii kuvanmuodostusohjelmalisäkkeelle määritetyn pohja-arkkitehtuurin mukaisesti. Luotu kuvanmuodostusohjelmalisäke sisältää Direct3D-version 11 -rajapinnan komentoja hyödyntävän järjestelmän. Luotuun kuvanmuodostus-ohjelmalisäkkeeseen rakennettiin myös laitteiden ja näyttötilojen luettelointiin tarvittavat menetelmät.

Kuvassa 29 havainnollistettu näyttötilan valitsintyökala on kehitetty grafiikka-ohjelmointirajapintavalitsimen tavoin vain testauskäyttöön. Työkalusta on mahdollista selata järjestelmässä tällä hetkellä kiinni olevat näytönohjaimet sekä muita yleisiä ominaisuuksia, joita näytönohjain ja järjestelmän päänäyttö tukevat. Järjestelmään rakennettiin mahdollisuus tehdä näytönohjaimen ja näyttötilan valinta myös ilman valitsintyökalun käyttöä.



Kuva 29. Tartarus3D:n näyttötilan valitsintyökalu.

Resurssijärjestelmän ohjelmalisäketuen testaukseen käytetään pohja-arkkitehtuurin kehityksen ohessa rakennettua ObjImporter-ohjelmalisäkettä. ObjImporter mahdollistaa Wavefrontin obj-tiedostojen tuonnin Tartaruksen resurssijärjestelmään.

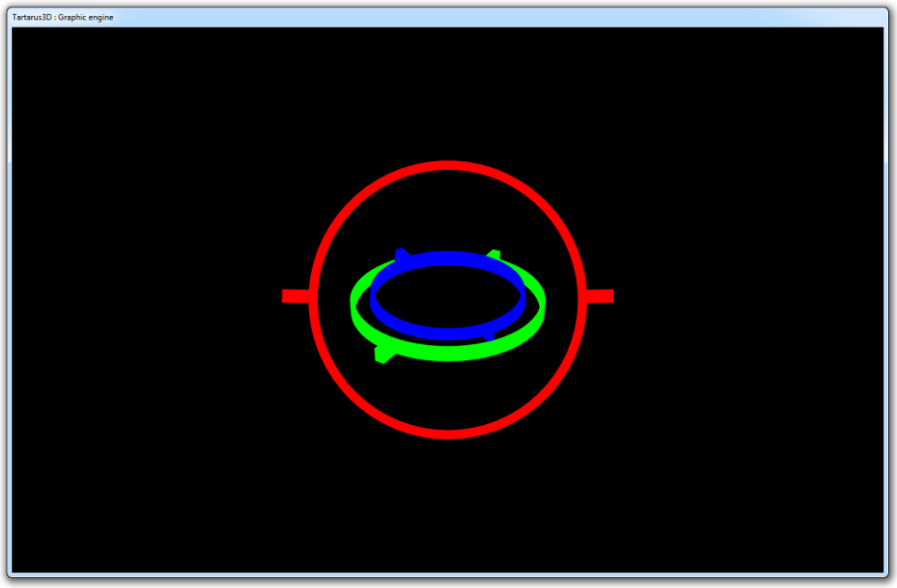
ObjImporter ei aluksi vaikuttanut toimivan kunnolla, koska järjestelmän alustava kuvanmuodostusyksikkö ei hyväksynyt geometriaobjektin verteksien rakennetta. Järjestelmän kuvanmuodostusyksikön ja resurssi-ohjelmalisäkkeiden käyttöön rakennettiin alustava versio dynaamisesta verteksirakenteesta, jonka jälkeen geometriaobjektien tuominen järjestelmään alkoi toimia oikein. Korjauksen jälkeen kuvan 30 mukainen Tartarus3D:n logo saatiin tuotua järjestelmään.

Kuva 30. Tartarus3D:n logo.

Näkymägraafin solmujärjestelmän laajentaminen testattiin rakentamalla alustavasti kuusi erilaista solmutyyppiä. Solmutyypit rekisteröidään solmujen rakentamisesta vastaavaan tehdasluokkaan cTartarus3D:n käynnistys-funktiossa. Tehdasluokka osaa rakentaa rekisteröinnin jälkeen halutunlaisia solmuja käyttämällä niiden rakennusfunktiota.

Resurssijärjestelmän tietovirta kuormittaa selvästi eniten CPU:ta. Resurssijärjestelmä joutuu huolehtimaan tiedostosta lähtöisin olevan tietovirran kääntämisen järjestelmän hyväksymään muotoon. Prosessi on hyvin hidas, mikäli luettavassa tiedostossa on useita tuhansia verteksimäärityksiä. Resurssien luontiprosessin aikaista tietovirran hitautta ei kuitenkaan tule suoraan verrata esimerkiksi näkymägraafin nopeuteen, sillä lukuprosessi tehdään yleensä ennen kuin objekteja hyödynnetään kuvanmuodostuksessa.

Kuvanmuodostus-, resurssi- ja näkymälohkot toimivat yhteistyössä kuitenkin niin, että ohjelma ei tunnu hidastelevan ajon-aikaisesti millään tavoin. 3d-malleja on mahdollista hallita näkymägraafin kehitettyjen tila- ja liikesolmujen avulla. Resurssijärjestelmän ansiosta samaan resurssiin on mahdollista viitata useaan otteeseen, kuten kuvassa 31. Tällöin on mahdollista käyttää myös saman geometriaresurssin verteksi- ja indeksipuskuria kuvanmuodostuksessa.



Kuva 31. Sama 3d-malli kolmella erilaisella tyylittelyllä.

Jatkokehityksessä tullaan vasta huomaamaan, miten kuvanmuodostuslohkon pohja-arkkitehtuuriratkaisu tulee toimimaan. Kuvanmuodostuslohkon toiminta monimutkaistuu huomattavasti efektimäärän kasvaessa. Myös näkymägraafin tietovirta tulee kasvamaan objektimäärän kasvaessa näkymässä. Näkymä-graafin solmutyyppien lukumäärä ei vaikuta grafiikkamoottorin reaaliaikaiseen tietovirtaan.

Näkymägraafin toimintaa on myös mahdollista hyödyntää erilaisten ohjelmistokokonaisuuksien parissa. Näkymägraafin tietojen tallennusmenetelmä antaa erinomaisen tavan tallentaa mitä tahansa tietoa, joka mahdollisesti halutaan olevan loogisesti sidoksissa toisiinsa. Näkymägraafin erilaisten läpikäyntimenetelmien avulla tieto on mahdollista hakea esille siinä järjestyksessä kuin graafia käyttävä ohjelma sitä tarvitsee.

Käytännössä koko pohja-arkkitehtuuria olisi mahdollista käyttää myös täysin erilaisten ohjelmistojen pohjana. Pohja-arkkitehtuurissa tulisi tällöin tehdä muutoksia ohjelmalisäkepalvelimiin sekä niihin sidoksissa oleviin rajapintoihin. Rajapintojen ohjelmalisäkekohtaiset komennot tulisi rakentaa sen mukaan millainen ohjelmisto pohja-arkkitehtuuria tulisi käyttämään.

Pohja-arkkitehtuurin käyttö olisi käytännössä hyvin helppo ottaa käyttöön sellaisen ohjelman pohjaksi, jossa on samantapaiset tietovirrat kuin grafiikkamoottoreissa. Jos tietovirrat pysyisivät samantapaisina, olisi mahdollista käyttää tietovirtoihin rakennettuja arkkitehtuurilohkoja ohjelmakohtaisella tavalla. Tällöin esimerkiksi resurssi-ohjelmalisäkkeet voisivat toimia vaikka tilastotietojen latausjärjestelmänä. Resurssijärjestelmän avulla tilastotietoihin olisi helppo päästä käsiksi mistä tahansa ohjelman osasta, jolloin myös tilastotietojen hallinnointi ja mahdollinen muokkaaminen olisi helposti toteutettavissa.

Näkymägraafin rakennetta on helppo muokata dynaamisen tehdasrakenteen avulla, sillä jokainen solmutyyppi olisi mahdollista luoda täysin ohjelman vaatimuksien mukaan. Graafin rakenne on mahdollista muokata sellaiseksi, että spatiaalisten suhteiden sijaan rakenne perustuisi loogisiin suhteisiin. Loogisten suhteiden avulla olisi mahdollista rakentaa tietojärjestelmä, jonka objektit voivat käytännössä sisältää mitä tahansa tietoa ohjelma tulee tarvitsemaan.

Pohja-arkkitehtuurissa ei käytännössä ole sidottuja ohjelmistolisäketyyppejä. Ohjelmistolisäketyyppejä on mahdollista lisätä ja poistaa sen mukaan mitä pohja-arkkitehtuuria käyttävä ohjelma niitä tarvitsee.

6 YHTEENVETO

Ohjelmalisäkkeiden avulla on mahdollista laajentaa järjestelmää ilman muutoksia sen lähdekoodiin. Kun järjestelmän kaikkia pohja-arkkitehtuurin osia on mahdollista laajentaa ulkoisilla ohjelmalisäkkeillä, on järjestelmä mahdollista rakentaa palasista sisältämään tarvittavat ominaisuudet.

Ohjelmalisäkkeiden rakentaminen on yksinkertaista, kun käytetään apuna rajapintaohjelmointia. Rajapinnat pakottavat luokat toteuttamaan vähintään ne luokat, joita järjestelmä tulee käyttämään ohjelmalisäkepalvelimen avulla. Rajapintojen avulla ohjelmalisäkkeet ovat aina yhteensopivia järjestelmän kanssa.

Ennen tässä työssä tehtyä arkkitehtuuria minulla ei ollut kokemusta nykyaikaisten grafiikkamoottorien pohja-arkkitehtuureista. Työn aikana perehtyminen vapaan lähdekoodin grafiikkamoottoreihin antoi hyvän kuvan nykyaikaisten grafiikkamoottorien arkkitehtuureista sekä niiden objektijärjestelmien toimintatavoista. Niiden hyväksi todettujen ominaisuuksien yhdistäminen alan kirjallisuuden tuoman tiedon kanssa mahdollisti tehokkaiden menetelmien valinnan Tartarus3D:n arkkitehtuuriin.

Grafiikkamoottorien pohja-arkkitehtuurin rakentamisella ei ole mitään varsinaista oikeaa tapaa. Työn aikana oli mahdollista huomata, että arkkitehtuuriset ratkaisut grafiikkamoottoreissa poikkeavat hyvinkin paljon toisistaan. Vaikka pääosat oli selvästi erotettavissa, oli grafiikkamoottoreissa kuitenkin eroja lohkojen sisäisten komponenttien toiminnassa.

Näkymägraafin hallinta on erittäin monimutkainen prosessi, jos sen haluaa rakentaa tehokkaaksi. Tartarus3D:hen alustavasti rakennetut näkymägraafin solmut antoivat hyvän kuvan, miten paljon pienikin optimointi voi vaikuttaa positiivisesti graafin muodostamalle sisäiselle tietovirralle. Näkymägraafi oli ennen tätä työtä itselleni täysin uusi tietorakenne, ja sen tehokkuus teki minuun suuren vaikutuksen.

Grafiikkamoottoreissa korostuu, kuinka tärkeää on rakentaa tehokkaita tietorakenteita. Järjestelmään luotuun apukirjastoon määritettiin tietorakenteita ja matemaattisia objekteja. Peliteollisuudessa käytettävien tietorakenteiden tutkiminen niin kirjallisuuden kuin Internetin avulla antoi erittäin paljon vahvistusta omaan tietorakennetietämykseen.

Osa Tartarus3D:ssä käytetyistä suunnittelumalleista oli jo entuudestaan minulle tuttuja. Muokattu versio abstraktista tehdas-suunnittelumallista oli itselleni uusi, ja sen opettelun jälkeen tuli heti mieleen useita paikkoja joihin olisin voinut käyttää sitä aiemmissa ohjelmointiprojekteissani. Myös tulevaisuudessa tulen melko varmasti hyödyntämään sen antamia mahdollisuuksia dynaamisten objektihallintajärjestelmien rakentamisessa.

Arkkitehtuurissa on paljon osia, joita voidaan parantaa käyttämällä kehittyneitä tekniikoita. Tekniikat on mahdollista toteuttaa samalla, kun pohja-arkkitehtuurin päälle rakennetaan varsinaista grafiikkamoottoria. Yksi parannusidea on nimeltään smart pointer -tekniikka, jonka avulla on mahdollista parantaa järjestelmän muistihallintaa ja estää mahdollisia muistivirheitä. Smart pointerit osaavat laskea viittaukset objekteihin ja tuhota ne vasta sitten, kun kaikki viittaukset niiden isäntäobjektiin on tuhottu. Lisäksi järjestelmä ei tällä hetkellä tue rinnakkaisajoa, joka saattaisi oikein toteutettuna nopeuttaa huomattavasti esimerkiksi resurssien latausta.

Pohja-arkkitehtuuri onnistui lopulta erinomaisesti ja pohja-arkkitehtuurin toiminta onnistui dynaamisen arkkitehtuurin vaatimuksien mukaisesti. Tartarus3D on siirtynyt jatkokehitysvaiheeseen FantasyCraftin tämänhetkiseen peliprojektiin.

LÄHTEET

[1] Morphologica - medical visualization, [www-dokumentti], Saatavilla: http://morphologica.net/w/, (Luettu 13.08.2011)

[2] David, L., Greg H., *How GPUs Work,* How things work, 2007, s. 1

[3] David, E., *Scene graphs and renderers,* 3D Game Engine Architecture, 2005, s. 149 - 152

[4] Stefan, Z., Oliver, D., *Applications, CPU limited or GPU limited*, 3D Game Engine Programming, 2004, s. 338 - 339.

[5] David, L., Greg, H., How GPUs Work, How things work, 2007, s. 5

[6] Allen, S., Wendy, J., *The components of DirectX 11*, Beginning DirectX 11 Game Programming, 2011, s. 7 - 10

[7] Mark, S., Kurt, A., *What is OpenGL Graphic System?*, The OpenGL Graphic System: A Specification (Version 4.2), 2011, s. 1 [pdf-dokumentti]
Saatavilla: http://www.opengl.org/registry/doc/glspec42.core.20110808.pdf (luettu 1.10.2011)

[8] Stefan, Z., Oliver, D., *API Independence through Interface Definitions*, 3D Game Engine Programming, 2004, s. 34 - 35

[9] Ron, P., *Graphs*, Data Structures for Game Programmers, 2003, s. 482

[10] David, E., *Node*, 3D Game Engine Architecture, 2005, s. 155 - 157

[11] Nikolaus, G., *Features*, Irrlicht - Lightning fast realtime 3d engine, [www-dokumentti], Saatavilla: http://irrlicht.sourceforge.net/features.html (Luettu 1.10.2011)

[12] Nikolaus, G., *Language Bindings*, Irrlicht - Ligtning fast realtime 3d engine, [www-dokumentti], Saatavilla: http://irrlicht.sourceforge.net/links.html (Luettu 1.8.2011)

[13] The Horde3D Team, *Horde3D Features*, Horde3D Next-generation graphics engine, [www-dokumentti], Saatavilla: http://www.horde3d.org/features.html (Luettu 10.8.2011)

[14] Erich, G., Richard, H., Ralph, J., John, V*., Factory Method*, Design Patterns: Elements of Reusable Object-Oriented Software, 1995, s. 107-117

[15] Torus Knot Software Ltd, *About*, Ogre, [www-dokumentti], saatavilla: http://www.ogre3d.org/about (Luettu 13.8.2011)

[16] Torus Knot Software Ltd, *Modules*, Ogre, [www-dokumentti], saatavilla: http://www.ogre3d.org/docs/api/html/modules.html (Luettu 13.8.2011)

[17] Markus, E., *Building a Better Plugin Architecture*, [www-dokumentti], saatavilla: http://www.nuclex.org/articles/5-cxx/4-building-a-better-plugin-architecture (Luettu: 1.9.2011)

[18] Microsoft, *Run-Time Dynamic Linking*, [www-dokumentti], saatavilla: http://msdn.microsoft.com/en-us/library/windows/desktop/ms685090(v=VS.85).aspx (Luettu 1.9.2011)

[19] Stefan, Z., Oliver, D., *What is an Interface?,* 3D Game Engine Programming, 2004, s. 41 - 42

[20] Microsoft, *Walkthrough: Creating and Using a Static Library (C++),* [www-dokumentti], saatavilla: http://msdn.microsoft.com/en-us/library/ms235627.aspx (Luettu 3.9.2011)

[21] Paul, P., *Tail for Win32, Tail for Win32 - Home Page*, [www-dokumentti], saatavilla: http://tailforwin32.sourceforge.net/ (Luettu 10.9.2011)

[22] David, A., Douglas, G., *Generic Programming in C++: Techniques*, [www-dokumentti], saatavilla: http://www.generic-programming.org/languages/cpp/techniques.php

[23] David, E*., Run-time type information*, 3D Game Engine Architecture, 2005, s. 105 - 111

[24] Erich, G., Richard, H., Ralph, J., John, V*., Singleton*, Design Patterns: Elements of Reusable Object-Oriented Software, 1995, s. 127 - 135

[26] Erich, G., Richard, H., Ralph, J., John, V*., Abstract Factory*, Design Patterns: Elements of Reusable Object-Oriented Software, 1995, s. 87 - 97